

**XML BASED SYSTEM FOR UPDATING A DOMAIN MODEL AND
GENERATING A FORMATTED OUTPUT**

Copyright Statement

A portion of the disclosure of this patent document
5 contains material which is subject to copyright protection.
The copyright owner has no objection to the facsimile
reproduction by anyone of the patent document or the patent
disclosure, as it appears in the Patent Office patent files
or records, but otherwise reserves all copyrights
10 whatsoever.

Technical Field

This invention relates, in general, to object oriented
programming systems, and more particularly to a system for
updating a domain model and generating a formatted output.

Background Art

Hyper Text Markup Language or HTML is synonymous with
the Internet and the World Wide Web (WWW). HTML allows
structural markup of WWW or 'Web' documents. XML or
eXtensible Markup Language is well known in the art as a
20 document markup language which offers human-readable
semantic markup and is also machine-readable. As a result,
XML provides the capability to create, parse and process
networked data.

XML documents are composed of entities, which are storage units containing text and/or binary data. Text is composed of character streams that form both the document's character data content and the document's meta-data markup. The markup describes the document's storage layout and logical structure. XML also provides a markup mechanism to impose constraints on the storage layout and logical structure of documents, and it provides mechanisms that can be used for strong typing.

In style and structure, XML documents look quite similar to HTML documents. However, when a Web server with XML content prepares data for transmission, the Web server must generate a context wrapper with each XML fragment, including pointers to an associated Document Type Definition (DTD) and one or more style sheets for formatting. Clients for the Web server that process XML must be able to unpack the content fragment, parse the fragment in the context according to the DTD (if needed), render (if needed) in accordance with the specified style sheet guidelines, and correctly interpret the hypertext semantics (e.g. links) associated with each of the different document tags. It is understood that a DTD is not required for an XML document, instead, the author can simply use an application-specific tagset. However, a DTD is useful because it allows applications to validate the tagset for proper usage. The DTD specifies the set of required and optional elements and their attributes for documents to conform to that type. In addition, the DTD specifies the names of the tags and the

relationships among elements in a document, for example,
nesting of elements.

One of the main issues during the development of XML
was expressing data stored in XML documents into various
5 formats. This has given rise to the development of
languages and standards such as XSL and XSLT from W3C. By
using an XSL style sheet one is able to create an HTML
representation of an XML document. Similarly, XSL style
sheets are used to transform XML documents into an HTML
10 representation. XSL implementations suffer two principal
limitations. First, XSLT requires the definition of rules
to transform specific types of XML elements into some other
type of XML like structure. Although the result may simply
comprise a stream, the navigation rules must be defined
15 based on types and the execution of the navigation rules is
based on the degree of uniqueness of the navigation. In
other words, the execution is not a simple procedural
operation. Secondly, XSLT has no formatting capability
which means that formatting is performed using flow/format
20 objects in XSL and is geared towards HTML forming.
Therefore, the XSL standards fall short for generating code
or non-XML output from a data model.

Another issue is concerned with the transformation of
XML data. Frequently, XML data needs to be transformed into
25 another valid XML format. This often comprises an update
activity not a creation activity. The updates are made to
specific pre-existing XML data, for example, converting
selected strings into an alternate format, which is not the

same as creating a new document. To be effective, the transformation needs to include an efficient navigation mechanism.

Accordingly, there remains a need for a mechanism for
5 navigating a data model and extracting specific data from the data model.

Summary of the Invention

30
35
40
45
50
55
60
65
70
75
80
85
90
95
100
105
110
115
120
125
130
135
140
145
150
155
160
165
170
175
180
185
190
195
200
205
210
215
220
225
230
235
240
245
250
255
260
265
270
275
280
285
290
295
300
305
310
315
320
325
330
335
340
345
350
355
360
365
370
375
380
385
390
395
400
405
410
415
420
425
430
435
440
445
450
455
460
465
470
475
480
485
490
495
500
505
510
515
520
525
530
535
540
545
550
555
560
565
570
575
580
585
590
595
600
605
610
615
620
625
630
635
640
645
650
655
660
665
670
675
680
685
690
695
700
705
710
715
720
725
730
735
740
745
750
755
760
765
770
775
780
785
790
795
800
805
810
815
820
825
830
835
840
845
850
855
860
865
870
875
880
885
890
895
900
905
910
915
920
925
930
935
940
945
950
955
960
965
970
975
980
985
990
995
1000
1005
1010
1015
1020
1025
1030
1035
1040
1045
1050
1055
1060
1065
1070
1075
1080
1085
1090
1095
1100
1105
1110
1115
1120
1125
1130
1135
1140
1145
1150
1155
1160
1165
1170
1175
1180
1185
1190
1195
1200
1205
1210
1215
1220
1225
1230
1235
1240
1245
1250
1255
1260
1265
1270
1275
1280
1285
1290
1295
1300
1305
1310
1315
1320
1325
1330
1335
1340
1345
1350
1355
1360
1365
1370
1375
1380
1385
1390
1395
1400
1405
1410
1415
1420
1425
1430
1435
1440
1445
1450
1455
1460
1465
1470
1475
1480
1485
1490
1495
1500
1505
1510
1515
1520
1525
1530
1535
1540
1545
1550
1555
1560
1565
1570
1575
1580
1585
1590
1595
1600
1605
1610
1615
1620
1625
1630
1635
1640
1645
1650
1655
1660
1665
1670
1675
1680
1685
1690
1695
1700
1705
1710
1715
1720
1725
1730
1735
1740
1745
1750
1755
1760
1765
1770
1775
1780
1785
1790
1795
1800
1805
1810
1815
1820
1825
1830
1835
1840
1845
1850
1855
1860
1865
1870
1875
1880
1885
1890
1895
1900
1905
1910
1915
1920
1925
1930
1935
1940
1945
1950
1955
1960
1965
1970
1975
1980
1985
1990
1995
2000
2005
2010
2015
2020
2025
2030
2035
2040
2045
2050
2055
2060
2065
2070
2075
2080
2085
2090
2095
2100
2105
2110
2115
2120
2125
2130
2135
2140
2145
2150
2155
2160
2165
2170
2175
2180
2185
2190
2195
2200
2205
2210
2215
2220
2225
2230
2235
2240
2245
2250
2255
2260
2265
2270
2275
2280
2285
2290
2295
2300
2305
2310
2315
2320
2325
2330
2335
2340
2345
2350
2355
2360
2365
2370
2375
2380
2385
2390
2395
2400
2405
2410
2415
2420
2425
2430
2435
2440
2445
2450
2455
2460
2465
2470
2475
2480
2485
2490
2495
2500
2505
2510
2515
2520
2525
2530
2535
2540
2545
2550
2555
2560
2565
2570
2575
2580
2585
2590
2595
2600
2605
2610
2615
2620
2625
2630
2635
2640
2645
2650
2655
2660
2665
2670
2675
2680
2685
2690
2695
2700
2705
2710
2715
2720
2725
2730
2735
2740
2745
2750
2755
2760
2765
2770
2775
2780
2785
2790
2795
2800
2805
2810
2815
2820
2825
2830
2835
2840
2845
2850
2855
2860
2865
2870
2875
2880
2885
2890
2895
2900
2905
2910
2915
2920
2925
2930
2935
2940
2945
2950
2955
2960
2965
2970
2975
2980
2985
2990
2995
3000
3005
3010
3015
3020
3025
3030
3035
3040
3045
3050
3055
3060
3065
3070
3075
3080
3085
3090
3095
3100
3105
3110
3115
3120
3125
3130
3135
3140
3145
3150
3155
3160
3165
3170
3175
3180
3185
3190
3195
3200
3205
3210
3215
3220
3225
3230
3235
3240
3245
3250
3255
3260
3265
3270
3275
3280
3285
3290
3295
3300
3305
3310
3315
3320
3325
3330
3335
3340
3345
3350
3355
3360
3365
3370
3375
3380
3385
3390
3395
3400
3405
3410
3415
3420
3425
3430
3435
3440
3445
3450
3455
3460
3465
3470
3475
3480
3485
3490
3495
3500
3505
3510
3515
3520
3525
3530
3535
3540
3545
3550
3555
3560
3565
3570
3575
3580
3585
3590
3595
3600
3605
3610
3615
3620
3625
3630
3635
3640
3645
3650
3655
3660
3665
3670
3675
3680
3685
3690
3695
3700
3705
3710
3715
3720
3725
3730
3735
3740
3745
3750
3755
3760
3765
3770
3775
3780
3785
3790
3795
3800
3805
3810
3815
3820
3825
3830
3835
3840
3845
3850
3855
3860
3865
3870
3875
3880
3885
3890
3895
3900
3905
3910
3915
3920
3925
3930
3935
3940
3945
3950
3955
3960
3965
3970
3975
3980
3985
3990
3995
4000
4005
4010
4015
4020
4025
4030
4035
4040
4045
4050
4055
4060
4065
4070
4075
4080
4085
4090
4095
4100
4105
4110
4115
4120
4125
4130
4135
4140
4145
4150
4155
4160
4165
4170
4175
4180
4185
4190
4195
4200
4205
4210
4215
4220
4225
4230
4235
4240
4245
4250
4255
4260
4265
4270
4275
4280
4285
4290
4295
4300
4305
4310
4315
4320
4325
4330
4335
4340
4345
4350
4355
4360
4365
4370
4375
4380
4385
4390
4395
4400
4405
4410
4415
4420
4425
4430
4435
4440
4445
4450
4455
4460
4465
4470
4475
4480
4485
4490
4495
4500
4505
4510
4515
4520
4525
4530
4535
4540
4545
4550
4555
4560
4565
4570
4575
4580
4585
4590
4595
4600
4605
4610
4615
4620
4625
4630
4635
4640
4645
4650
4655
4660
4665
4670
4675
4680
4685
4690
4695
4700
4705
4710
4715
4720
4725
4730
4735
4740
4745
4750
4755
4760
4765
4770
4775
4780
4785
4790
4795
4800
4805
4810
4815
4820
4825
4830
4835
4840
4845
4850
4855
4860
4865
4870
4875
4880
4885
4890
4895
4900
4905
4910
4915
4920
4925
4930
4935
4940
4945
4950
4955
4960
4965
4970
4975
4980
4985
4990
4995
5000
5005
5010
5015
5020
5025
5030
5035
5040
5045
5050
5055
5060
5065
5070
5075
5080
5085
5090
5095
5100
5105
5110
5115
5120
5125
5130
5135
5140
5145
5150
5155
5160
5165
5170
5175
5180
5185
5190
5195
5200
5205
5210
5215
5220
5225
5230
5235
5240
5245
5250
5255
5260
5265
5270
5275
5280
5285
5290
5295
5300
5305
5310
5315
5320
5325
5330
5335
5340
5345
5350
5355
5360
5365
5370
5375
5380
5385
5390
5395
5400
5405
5410
5415
5420
5425
5430
5435
5440
5445
5450
5455
5460
5465
5470
5475
5480
5485
5490
5495
5500
5505
5510
5515
5520
5525
5530
5535
5540
5545
5550
5555
5560
5565
5570
5575
5580
5585
5590
5595
5600
5605
5610
5615
5620
5625
5630
5635
5640
5645
5650
5655
5660
5665
5670
5675
5680
5685
5690
5695
5700
5705
5710
5715
5720
5725
5730
5735
5740
5745
5750
5755
5760
5765
5770
5775
5780
5785
5790
5795
5800
5805
5810
5815
5820
5825
5830
5835
5840
5845
5850
5855
5860
5865
5870
5875
5880
5885
5890
5895
5900
5905
5910
5915
5920
5925
5930
5935
5940
5945
5950
5955
5960
5965
5970
5975
5980
5985
5990
5995
6000
6005
6010
6015
6020
6025
6030
6035
6040
6045
6050
6055
6060
6065
6070
6075
6080
6085
6090
6095
6100
6105
6110
6115
6120
6125
6130
6135
6140
6145
6150
6155
6160
6165
6170
6175
6180
6185
6190
6195
6200
6205
6210
6215
6220
6225
6230
6235
6240
6245
6250
6255
6260
6265
6270
6275
6280
6285
6290
6295
6300
6305
6310
6315
6320
6325
6330
6335
6340
6345
6350
6355
6360
6365
6370
6375
6380
6385
6390
6395
6400
6405
6410
6415
6420
6425
6430
6435
6440
6445
6450
6455
6460
6465
6470
6475
6480
6485
6490
6495
6500
6505
6510
6515
6520
6525
6530
6535
6540
6545
6550
6555
6560
6565
6570
6575
6580
6585
6590
6595
6600
6605
6610
6615
6620
6625
6630
6635
6640
6645
6650
6655
6660
6665
6670
6675
6680
6685
6690
6695
6700
6705
6710
6715
6720
6725
6730
6735
6740
6745
6750
6755
6760
6765
6770
6775
6780
6785
6790
6795
6800
6805
6810
6815
6820
6825
6830
6835
6840
6845
6850
6855
6860
6865
6870
6875
6880
6885
6890
6895
6900
6905
6910
6915
6920
6925
6930
6935
6940
6945
6950
6955
6960
6965
6970
6975
6980
6985
6990
6995
7000
7005
7010
7015
7020
7025
7030
7035
7040
7045
7050
7055
7060
7065
7070
7075
7080
7085
7090
7095
7100
7105
7110
7115
7120
7125
7130
7135
7140
7145
7150
7155
7160
7165
7170
7175
7180
7185
7190
7195
7200
7205
7210
7215
7220
7225
7230
7235
7240
7245
7250
7255
7260
7265
7270
7275
7280
7285
7290
7295
7300
7305
7310
7315
7320
7325
7330
7335
7340
7345
7350
7355
7360
7365
7370
7375
7380
7385
7390
7395
7400
7405
7410
7415
7420
7425
7430
7435
7440
7445
7450
7455
7460
7465
7470
7475
7480
7485
7490
7495
7500
7505
7510
7515
7520
7525
7530
7535
7540
7545
7550
7555
7560
7565
7570
7575
7580
7585
7590
7595
7600
7605
7610
7615
7620
7625
7630
7635
7640
7645
7650
7655
7660
7665
7670
7675
7680
7685
7690
7695
7700
7705
7710
7715
7720
7725
7730
7735
7740
7745
7750
7755
7760
7765
7770
7775
7780
7785
7790
7795
7800
7805
7810
7815
7820
7825
7830
7835
7840
7845
7850
7855
7860
7865
7870
7875
7880
7885
7890
7895
7900
7905
7910
7915
7920
7925
7930
7935
7940
7945
7950
7955
7960
7965
7970
7975
7980
7985
7990
7995
8000
8005
8010
8015
8020
8025
8030
8035
8040
8045
8050
8055
8060
8065
8070
8075
8080
8085
8090
8095
8100
8105
8110
8115
8120
8125
8130
8135
8140
8145
8150
8155
8160
8165
8170
8175
8180
8185
8190
8195
8200
8205
8210
8215
8220
8225
8230
8235
8240
8245
8250
8255
8260
8265
8270
8275
8280
8285
8290
8295
8300
8305
8310
8315
8320
8325
8330
8335
8340
8345
8350
8355
8360
8365
8370
8375
8380
8385
8390
8395
8400
8405
8410
8415
8420
8425
8430
8435
8440
8445
8450
8455
8460
8465
8470
8475
8480
8485
8490
8495
8500
8505
8510
8515
8520
8525
8530
8535
8540
8545
8550
8555
8560
8565
8570
8575
8580
8585
8590
8595
8600
8605
8610
8615
8620
8625
8630
8635
8640
8645
8650
8655
8660
8665
8670
8675
8680
8685
8690
8695
8700
8705
8710
8715
8720
8725
8730
8735
8740
8745
8750
8755
8760
8765
8770
8775
8780
8785
8790
8795
8800
8805
8810
8815
8820
8825
8830
8835
8840
8845
8850
8855
8860
8865
8870
8875
8880
8885
8890
8895
8900
8905
8910
8915
8920
8925
8930
8935
8940
8945
8950
8955
8960
8965
8970
8975
8980
8985
8990
8995
9000
9005
9010
9015
9020
9025
9030
9035
9040
9045
9050
9055
9060
9065
9070
9075
9080
9085
9090
9095
9100
9105
9110
9115
9120
9125
9130
9135
9140
9145
9150
9155
9160
9165
9170
9175
9180
9185
9190
9195
9200
9205
9210
9215
9220
9225
9230
9235
9240
9245
9250
9255
9260
9265
9270
9275
9280
9285
9290
9295
9300
9305
9310
9315
9320
9325
9330
9335
9340
9345
9350
9355
9360
9365
9370
9375
9380
9385
9390
9395
9400
9405
9410
9415
9420
9425
9430
9435
9440
9445
9450
9455
9460
9465
9470
9475
9480
9485
9490
9495
9500
9505
9510
9515
9520
9525
9530
9535
9540
9545
9550
9555
9560
9565
9570
9575
9580
9585
9590
9595
9600
9605
9610
9615
9620
9625
9630
9635
9640
9645
9650
9655
9660
9665
9670
9675
9680
9685
9690
9695
9700
9705
9710
9715
9720
9725
9730
9735
9740
9745
9750
9755
9760
9765
9770
9775
9780
9785
9790
9795
9800
9805
9810
9815
9820
9825
9830
9835
9840
9845
9850
9855
9860
9865
9870
9875
9880
9885
9890
9895
9900
9905
9910
9915
9920
9925
9930
9935
9940
9945
9950
9955
9960
9965
9970
9975
9980
9985
9990
9995
10000
10005
10010
10015
10020
10025
10030
10035
10040
10045
10050
10055
10060
10065
10070
10075
10080
10085
10090
10095
10100
10105
10110
10115
10120
10125
10130
10135
10140
10145
10150
10155
10160
10165
10170
10175
10180
10185
10190
10195
10200
10205
10210
10215
10220
10225
10230
10235
10240
10245
10250
10255
10260
10265
10270
10275
10280
10285
10290
10295
10300
10305
10310
1

directives, a user can write a procedural sequence of instructions to quickly extract data from a XML source data model (e.g. an XML document) and format the extracted data into a desired stream for a target data model. The target data model may comprise a simple ASCII file, or an HTML file, or generated content of some other file format. The mechanism according to the invention provides a target data model without restrictions.

According to another aspect of the invention, the mechanism, i.e. directives, is implemented utilizing XML thereby providing a language interface which is natural to most users.

In a first aspect, the present invention provides a mechanism for manipulating information in a source data model and creating a target data model, the mechanism includes, (a) a template module having a directive to manipulate selected data in the source data model; (b) a template processing module to process the directive contained in the template module; and (c) the template processing module further includes a component to generate a DOM tree for navigating the template module to manipulate said source data model.

In another aspect, the present invention provides a method for manipulating selected data in a source data model, the method comprises the steps of: (a) defining a template file having a directive specifying the data to be extracted from the source data model; (b) generating a DOM

tree for navigating the template file; and (c) navigating the template file and applying the directive to manipulate selected data in the source data model.

5 In yet another aspect, the present invention provides a computer program product for an application program for creating objects, the application program includes a utility for manipulating information in a source data model and creating a target data model, the computer program product comprises: a recording medium; means recorded on the medium
10 for instructing a computer to perform the steps of, (a) defining a template file having a directive specifying the data to be extracted from the source data model; (b) generating a DOM tree for navigating the template file; and (c) navigating the template file and applying the directive
15 to manipulate selected data in the source data model.

Additional features and advantages are realized through the techniques of the present invention. Other embodiments and aspects of the invention are described in detail herein and are considered a part of the claimed invention.

20

Brief Description of the Drawings

25

The subject matter which is regarded as the invention is particularly pointed out and distinctly claimed in the claims at the conclusion of the specification. The foregoing and other objects, features, and advantages of the invention are apparent from the following detailed

description taken in conjunction with the accompanying drawings in which:

5 FIG. 1 shows in diagrammatic form a system for updating a domain model and generating a formatted output in accordance with the present invention;

FIG. 2 shows in flowchart form the method steps for processing directives according to the present invention;

10 FIG. 3 shows in flowchart form the method steps for executing directives according to the present invention;

FIG. 4 shows in flowchart form the method steps for resolving a macro according to the present invention;

15 FIG. 5 shows in flowchart form the method steps for resolving scoping according to the present invention;

20 FIG. 6 shows in diagrammatic form a formatted output generated according to the present invention; and

FIG. 7 shows in diagrammatic form a domain model manipulation example according to the present invention.

Best Mode for Carrying Out the Invention

Reference is first made to Fig. 1 which shows in diagrammatic form a mechanism according to the present invention for updating a domain model and generating a formatted output. The mechanism comprises a template-driven implementation. In the preferred embodiment, the mechanism is in an application development program for creating objects, etc., according to object-oriented principles and the domain models and trees define an object model.

The mechanism is indicated generally by reference 10 and comprises a template file or module 12 and a template driven emitter or template processing module 14. As will be described in more detail below, the template driven emitter 14 applies the template file 12 to a domain model 16 to produce a generated output file 18. In the context of the present invention, the domain model 16 comprises a source data model and the generated output file 18 comprises a target data model. The source data model 16 contains read-only data which is extracted by the mechanism 10 and used to generate a formatted output, i.e. the target data model 18. The target data model 18 contains both read/write data that is manipulated by the mechanism 10. According to the invention, the mechanism 10 provides the capability to navigate the data models 16, 18 and also manipulate the target data model 18.

The template file 12 is created as a text file with directives that navigate the source data file 16 and navigate and manipulate the target data file 18. While the mechanism 10 is described in the context of an XML based data model 16, the mechanism 10 is not concerned with the structure of the model 16. The template file 12 is created using a conventional editor. The mechanism 10 only responds to navigational, substitution and emit directives contained in the template files 12. As will be described in more detail below, the data model 16 is traversed in a symbolic fashion, i.e. naming the object relationships as strings, rather than direct object references. The attribute values are retrieved in the same fashion, i.e. by naming the attribute in a string.

In the preferred embodiment, the template file 12 is expressed in XML format and comprises directives and macros. The specification of the document type definition (DTD) for the template file 12 is provided below in Appendix I. The directives are commands or rules that define the flow of control. The macros provide for string manipulation. The mechanism 10 includes domain model navigation directives, domain model manipulation directives, output manipulation directives, logical operation directives, and code section directives. The domain model navigation directives are used to navigate the domain model (i.e. the source data model 16). The domain model manipulation directives are used to modify the domain model (i.e. target data model 18). The output manipulation directives are used to modify the generated output in the target data model 18. The logical

operation directives are used to change the logical processing of a template file 12. The code section directives are used to maintain code sections in the generated output (i.e. target data model 18). A full
5 listing of the directives is provided below in Appendix II.

As described above, the source data model (i.e. domain model 16 in Fig. 1) contains read-only data that the mechanism 10 can extract and use to generate a formatted output (i.e. the target data model 18). A feature of the
10 mechanism 10 according to the present invention is the ability to manipulate the target data model 18. According to this aspect of the invention, the target data model 18 contains read/write data which can be manipulated according to directives contained in the template file 12. A full
15 listing of the directives is provided below in Appendix II, and directives for manipulating the target data model 18 include *updatetargetscope*, *targetscope*, *addtargetscope*, and *removetargetscope*.

The *updatetargetscope* directive is used to update an
20 element of a DOM tree associated with a domain model. If the element does not exist it is inserted into the DOM tree. The syntax for the *updatetargetscope* directive is shown below in Appendix II. The following example describes the operation of the *updatetargetscope* directive:

25 Suppose the model defines an element "Class" which has two other embedded elements. This document (i.e the source data file 16) is called "class.xml".

class.xml
 <Class name="Set">
 <Method name="add">
 5 </Method>
 <Method name="del">
 </Method>
 </Class>

10 Also consider the following template (i.e. template
 file 12):

<?xml version="1.0"?>
 <!DOCTYPE _TDEBlock_ SYSTEM "dtd/tde.dtd">
 <_TDEBlock_ DOMTree="class.xml">
 15 <define MACRO=newMethodName>addNew</define>
 <targetscope NAME="Method">
 <updatetargetscope NAME="name"
 TYPE="ATTRIBUTE ">\$newMethodName\$</updatetargetscope>
 </targetscope>
 20 </_TDEBlock_>

The resulting DOM tree when the template is applied to
 the document "class.xml" is as follows:

<Class name="Set">
 <Method name="addNew">
 25 </Method>
 <Method name="del">
 </Method>
 </Class>

It will be appreciated that the name of the first method changes from "add" to "addNew".

5 The targetscope directive is used to navigate to an element that is part of an existing DOM tree associated with. The syntax for the targetscope directive is described below in Appendix II. The following example illustrates the operation of the targetscope directive:

10 Suppose the model defines an element "Class" which has two other embedded elements. This document (i.e. the source data file 16) is called "class.xml".

15 class.xml
<Classlist>
<Class name="Set">
<Method name="add">
</Method>
<Method name="del">
</Method>
</Class>
</Classlist>

20 Also consider the following template (i.e. the template file 12):

25 <?xml version="1.0"?>
<!DOCTYPE _TDEBlock_ SYSTEM "dtd/tde.dtd">
<_TDEBlock_ DOMTree="class.xml">
<targetscope NAME="Class">
</targetscope>

</_TDEBlock_>

The template driven emitter 14 navigates to the Class element in the class.xml document. It will be understood that no emitted output is created since this directive just
5 navigates the target data model 18 (i.e. the target document). In this case the targeted XML document is class.xml.

The addtargetscope directive is used to insert an element into a DOM tree associated with a domain model. The
10 syntax for the addtargetscope directive is described below in Appendix II. The following example illustrates the operation of the addtargetscope directive.

Suppose the model defines an element 'Class' which has two other embedded elements. This document (i.e. the source
15 data file 16) is called "class.xml".

```
class.xml
<Classlist>
  <Class name="Set">
    <Method name="add">
20    </Method>
    <Method name="del">
    </Method>
  </Class>
</Classlist>
```

Also consider the following template (i.e. the template file 12):

```
5      <?xml version="1.0"?>
      <!DOCTYPE _TDEBlock_ SYSTEM "dtd/tde.dtd">
      <_TDEBlock_ DOMTree="class.xml">
      <targetscope NAME="Class">
      <addtargetscope NAME="Method">
      <addtargetscope NAME="Argument"></addtargetscope>
      </addtargetscope>
10     </targetscope>
      </_TDEBlock_>
```

The resulting DOM tree when the template is applied to the document "class.xml" is as follows:

```
15     <ClassList>
      <Class name="Set">
      <Method>
      <Argument/>
      </Method>
      <Method name="add">
20     </Method>
      <Method name="del">
      </Method>
      </Class>
      </Classlist>
```

25 The removetargetscope directive is used to delete an element from a DOM tree associated with a domain model. The syntax for the removetargetscope directive is shown below in

Appendix II. The following example describes the operation of the removetargetscope directive:

Suppose the model defines an element "Class" which has two other embedded elements. This document (i.e. the source data file 16) is called "class.xml".

```
class.xml
<Class name="Set">
  <Method name="add">
  </Method>
  <Method name="del">
  </Method>
</Class>
```

Also consider the following template (i.e. the template file 12):

```
<?xml version="1.0"?>
<!DOCTYPE _TDEBlock_ SYSTEM "dtd/tde.dtd">
<_TDEBlock_ DOMTree="class.xml">
  <removetargetscope NAME="Method"/>
</_TDEBlock_>
```

The result of when the template is applied to the document "class.xml" is as follows:

```
<Class name="Set">
  <Method name="del">
  </Method>
</Class>
```

It will be appreciated that the first method element was removed. Since no index was specified the default value for the index is 0.

5 It will be appreciated that these directives are very useful in applications, such as "Web" applications and database applications, which require updating of a model stored in XML or a model stored in a database. The other directives included in the mechanism 10 according to the present invention are described below in Appendix II.

10 The mechanism 10 also includes XML specific directives such as *updatetargetdoctype*. These XML specific directives are for working with data models that are stored as XML (unlike other directives which are not XML data model specific).

15 The *updatetargetdoctype* directive is used to update the !DOCTYPE element of a DOM tree associated with a domain model. If the element does not exist it is inserted into the DOM tree. The syntax of the *updatetargetdoctype* is described in Appendix II below. The operation of the
20 *updatetargetdoctype* is illustrated by the following example which involves adding a simple !DOCTYPE element (additional examples are provided in Appendix II).

Consider the following template file:

25 <?xml version="1.0"?>
<!DOCTYPE _TDEBlock_ SYSTEM "../.../.../dtd/tde.dtd">
<_TDEBlock_>


```

<updatetargetdoctype PUBLIC_NAME="Chemistry"
PUBLIC_URL="http://sunsite.unc.edu/public/chemistry.dtd"
ROOT_ELEMENT_NAME="myroot"/>
</_TDEBlock_>

```

5

The result of when the template is applied is as follows:

```

<?xml version="1.0" standalone="no"?>
<!DOCTYPE myroot PUBLIC "Chemistry"
"http://sunsite.unc.edu/public/chemistry.dtd">

```

10

The mechanism 10 according to the present invention also includes macro transformations. A macro associates a name with a set of values and enables string substitution within the template file 12. The macro transformations are useful in situations that require text to be changed to uppercase, lowercase, truncated, etc. The following macro transformations are provided in the mechanism 10: *toupper*, *tolower*, *truncate*, *separate*, *strip*, *stripTrailing*, *stripLeading*, *change in_pattern out_pattern*, *numWords*, *words[i]*, and *word[i]*.

15

20

The *toupper* macro converts the macro value to all upper case letters. If an optional parameter 'i' (as described in Appendix II) is specified, then only the i'th character is converted to upper case. The *tolower* macro converts the macro value to all lower case letters. If the optional parameter 'i' is specified in that case, then only the i'th character is converted to lower case. The *truncate* macro truncates the macro value to a specified parameter 'maxlen'

25

characters long by removing vowels. If no maxlen parameter is specified, then the macro truncates to the default of 8 characters. The separate macro separates the values in a multivalue macro by separator. If the separator is a string/character, the separator must be enclosed by quotes 5 '' . Otherwise it is assumed to be a macro name and the value of that macro is used as the separator. It is to be noted that when used in a <repeat> statement, repeat blocks are separated by a separator. The strip macro removes all leading and trailing white space characters from the macro 10 value. If a string pattern is specified (either as a string or a macro name), the pattern will be stripped off from the macro value. The stripTrailing macro removes all trailing white space characters from the macro value. An optional pattern to strip may be specified. The stripLeading macro 15 removes all leading white space characters from the macro value. An optional pattern to strip may be specified. The change in_pattern out_pattern macro replaces every occurrence of the in_pattern in the macro value with out_pattern. Strings/characters must be enclosed by quotes, 20 otherwise the parameter is assumed to be a macro name and the value of that macro will be used. The increment macro increments the value of the macro by one. If an optional parameter 'i' is specified, then the increment step is defined by 'i'. The numWords macro returns the number of 25 words in a string. For example, if the string contains "This is an apple pie", then the macro numWords returns '5'. The words[i] macro returns the string after the i'th word inclusively. For the previous example, words [3] would return "an apple pie". The word[i] macro returns the ith 30

word in a string. For the previous example, the macro word[3] would return "an". The macro transformations included in the mechanism 10 according to the present invention are further described in Appendix II.

5 According to another aspect, the mechanism 10 utilizes a "tree" navigation scheme to perform transformations in the source data model 16 (and the target data model 18). As will be described in more detail below, to apply a transformation to a node in the data model 16 or 18, the
10 mechanism 10 first navigates the root node for the data model, then down to the child node until the specific node of interest is located. Once located, the transformation rule(s) is applied to the node. As tree navigation is not data model specific, the mechanism 10 according to the
15 present invention is advantageously flexible and can support various data model types other than XML.

 The mechanism 10 also includes a facility for code sections. The code sections are analogous to methods and have application for reusing templates, or for overriding
20 other code sections with the same name. The code sections comprise the directives *code* and *call*, which are described in more detail in Appendix II.

 Reference is next made to Figs. 2 to 5 which show the method steps embodied in processing the template file 12
25 according to the present invention.

Fig. 2 shows in flow chart form the method steps embodied in the mechanism 10 for processing directives according to the present invention. The method steps comprise a process which is indicated generally by reference 100 in Fig. 2. As described above, the directives for navigating and/or manipulating the data model are contained in the template file 12 (Fig. 1), and the template file 12 is processed by the template driven emitter 14 (Fig. 1). The first step as indicated by block 110 in Fig. 2 involves initializing the template driven emitter 14, which in the preferred embodiment is capable of processing XML documents and templates. Initialization of the template driven emitter 14 includes the following operations: creating and populating a macro table with predefined macros (which will be used for string substitution in the template file) and creating a code section table (which will be populated with code sections, defined by name using the code directive, that will be emitted on the occurrence of a call directive using the code sections' name(s)). Additionally, the template driven emitter generates a DOM tree for the template file and for the source data model, both trees used for navigating respectively the template file and the source data model. DOM or document object model is an API for HTML and XML documents established through the W3C Organization. DOM defines the logical structure of documents and the way a document is accessed and manipulated. See <http://www.w3.org/dom> for more information. As used herein, DOM trees are understood as used in the context of the DOM specification but need strictly follow the DOM specification. Indeed, as used herein, DOM trees could be

trees according to conventional understandings or defined according to other non-DOM specifications.

5 The next step (block 112) in the process 100 involves getting the first or next directive from the template file 12. As described above, the template file 12 is created as a text file with the directives specified according to the definitions contained in Appendix II. The next step in decision block 114 involves ascertaining whether the directive is a directive for the template driven emitter 14. (It will be appreciated that only template driven emitter directives can drive the operation of the template driven emitter 14.) If the directive retrieved in step 112 is not a template driven emitter (TDE) directive, then a check (decision block 128) is made to determine if there any further directives in the template file 12. If no, then the process 100 for processing the template file 12 is terminated. If yes, then the next directive in the template file 12 is retrieved (step 112) and a check is made to ascertain if the directive is a TDE directive (decision block 114).

Referring to Fig. 2, if the directive is a TDE directive (as determined in decision block 114), then the next step (block 116) in the process 100 involves parsing the content of the directive for any macros which may be included. If the content does include a macro value (decision block 118), then a process 300 for resolving the macro name is invoked in the block 120. The result of the macro name process 300 is a macro value for the macro name.

The macro name resolving process 300 is described below with
 reference to Fig. 4. As shown in Fig. 2, the next step in
 the process 100 involves determining if the macro value
 returned from the process 300 requires scoping (decision
 5 block 122). If the macro value requires scoping, then a
 process 400 for scoping the macro value is invoked in block
 130. The macro value scoping process 400 is described in
 more detail below with reference to Fig. 5. Once the macro
 value has been scoped, the directive is executed as
 10 indicated by block 124 in Fig. 2. If the macro value does
 not require scoping as determined in decision block 122,
 then the next step is also executing the directive in block
 124. If the directive does not contain a macro value as
 determined in decision block 118, then processing also
 15 proceeds to executing the directive in block 124. In block
 124, the directive is executed 200 as will be described in
 more detail with reference to Fig. 3 below. The result of
 executing the directive, i.e. content, is written to an
 output buffer as indicated in block 126. Next, a check is
 20 made in decision block 128 to ascertain if there any more
 directives in the template file 12. If there are more
 directives in the template file 12, then the processing
 steps starting at block 112 are repeated as described above.
 If there are no more directives, then the processing of the
 25 template file 12 is completed and the process 100 ends.

The processing of the template file 12 (Fig. 1) is
 implemented in the template driven emitter 14 (Fig. 1) as
 will be apparent to one skilled in the art based on the
 foregoing description.

Reference is next made to Fig. 3, which shows the method steps embodied in a process for executing the directive as indicated by block 124 (Fig. 2). As shown in Fig. 3, the process for executing the directives is

5 indicated generally by reference 200. The directives are processed according the definitions found in Appendix II. The first operation in the process 200 involves pushing the current directive context onto a stack as indicated in block 210. Next, the start tag of the directive is processed

10 (block 212) by, among other things, parsing the directive and performing the steps required to execute the action of the directive in accordance with the relevant definitions found in Appendix II. It should be apparent to those skilled in the art how to implement the specifications and definitions of Appendix II. The next operation involves

15 determining if the directive contains any 'children' in decision block 214. If there is a child, then the current context is set to the child directive in block 216, and pushed onto the stack in block 210. The steps in blocks 212 and 214 are repeated in order to navigate the DOM tree.

20

Referring to Fig. 3, if the directive does not contain a child or any more children (as determined in decision block 214), then the end tag of the directive is processed in block 218. The last operation in the directive processing

25 involves popping the current directive context from stack (which was pushed in block 210) to restore the state data, as indicated in block 220. After this step, the directive processing is ended.

Reference is next made to Fig. 4 which shows the method steps embodied in a process for resolving the macro name as indicated by block 120 (Fig. 2). As shown in Fig. 4, the process for resolving the macro name is indicated generally by reference 300. The first step in resolving the macro name process 300 involves ascertaining if the macro name is stored in the macro table, as indicated by decision block 310. If the macro name is contained in the macro table, then the value for the macro name is extracted from the macro table, as indicated in block 312, and the process 300 is completed. If the macro name is not stored in the macro table (decision block 310), then a check is made to ascertain if the macro name is stored in the domain model, as indicated in decision block 314. If the macro name is stored in the model, then the value for the macro name is extracted from the model, as indicated in block 316, and the process 300 for resolving the macro name is completed. If the macro name is not stored in the model (as determined in decision block 314), then the value for the macro name is set to the name encased by a '\$' character in the template file 12. After step 318, the process 300 is completed.

Reference is next made to Fig. 5, which shows the method steps embodied in a process for resolving the scoping of the macro name as indicated by block 130 (Fig. 2). As shown in Fig. 5, the process for resolving the scoping of the macro name, i.e. navigating the DOM tree, is indicated generally by reference 400. The first step in the process 400 for resolving the scope of the macro name involves

processing the root scope name in block 410. Next, a
determination is made in decision block 412 to determine if
the scope refers to a child node in the DOM tree for the
model. If the scope is a child node, then the scope context
5 is moved to the child node, as indicated in block 414. After
step 414, a check is made in decision block 416, to
determine if the scope name contains any additional scopes.
If there are additional scopes, then the scope name is
processed in block 410 and decision block 412 as described
10 above.

Referring to Fig. 5, if the scope name does not refer
to a child node (as determined in decision block 412), then
a check is made in decision block 418 to ascertain if the
scope name refers to a parent node in the DOM tree. If yes,
15 then the scope context is moved to the parent node as
indicated in block 420. After this step, a check is made to
ascertain if the scope name contains additional scopes in
decision block 416, as described above. If the scope name
does not correspond to the parent node (decision block 418),
20 then a check is made in decision block 422 to determine if
the scope name refers to the root node in the DOM tree. If
the scope name refers to the root node, then the scope
context is moved to the root node, as indicated by block
424. Next, a check is made in decision block 416 to
25 determine if the scope name contains additional scopes as
described above.

The operation of the mechanism 10 according to the present invention is further described with reference to two examples depicted in Fig. 6 and Fig. 7, respectively.

Reference is made to Fig. 6, which depicts a formatted
5 output file 28 generated from a source data file .26
according to a template file 22. The template file 22 is
processed according to a template driven emitter 24. The
arrangement of the template file 22 and the template driven
emitter 24 are as described above. In operation, the
10 template driven emitter 24 reads the source data file 26 and
creates a DOM tree. The template driven emitter 24 also
reads the template file 22 and creates a DOM tree (as
described above). When the template file 22 is first
processed, the template driven emitter 24 sets the current
15 context to the root element in both DOM trees. Subsequently,
navigation of the DOM tree is controlled by the scope
directive(s) contained in the template file 22. As shown in
Fig. 6, the template file 22 defines the root element as
'TDEBlock' (Line 3). In Line 4, the template file 22
20 includes a directive which defines the formatted output file
28 named by a macro value. Line 6 of the template file 22
includes a scope directive which changes the context (i.e.
navigates the DOM tree) to the 'Method'. As shown in Fig. 6,
the generated output in the output file 28 is indicated by
25 'void add()' in Line 2. The output file 28 is stored under
the name 'Set.java', which was defined by the 'outfile'
directive in the template file 22 (Line 4).

Reference is next made to Fig. 7, which shows an example of a DOM tree manipulation according to the present invention. As shown in Fig. 7, the template driven emitter 34 processes a template file 32 which modifies a source data file 36 to generate an output file 38. The output file 38 comprises the source file 36 associated with a modified DOM tree. As shown in Fig. 7, the template file 32 includes an 'addtargetscope' directive in Line 4 and another 'addtargetscope' directive in Line 5. As described above, the addtargetscope directive is used to insert an element into the DOM tree. For the template file 32, the first directive directs the addition of an element called 'Method' to the DOM tree. The second directive in the template file 32 directs the addition of an attribute 'name' to the element 'Method'. The element 'Method' with the attribute 'name' appears in Line 7 of the generated output file 38 as shown in Fig. 7. It will be appreciated that the data model is navigated in a symbolic manner, i.e. naming the object relationships as strings, rather than direct object references. Similarly, the attribute values are retrieved by naming the attribute string.

The detailed descriptions may have been presented in terms of program procedures executed on a computer or network of computers. These procedural descriptions and representations are the means used by those skilled in the art to most effectively convey the substance of their work to others skilled in the art. They may be implemented in hardware or software, or a combination of the two.

A procedure is here, and generally, conceived to be a self-consistent sequence of steps leading to a desired result. These steps are those requiring physical manipulations of physical quantities. Usually, though not necessarily, these quantities take the form of electrical or magnetic signals capable of being stored, transferred, combined, compared, and otherwise manipulated. It proves convenient at times, principally for reasons of common usage, to refer to these signals as bits, values, elements, symbols, characters, terms, numbers, objects, attributes or the like. It should be noted, however, that all of these and similar terms are to be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities.

Further, the manipulations performed are often referred to in terms, such as adding or comparing, which are commonly associated with mental operations performed by a human operator. No such capability of a human operator is necessary, or desirable in most cases, in any of the operations described herein which form part of the present invention; the operations are machine operations. Useful machines for performing the operations of the present invention include general purpose digital computers or similar devices.

Each step of the method may be executed on any general computer, such as a mainframe computer, personal computer or the like and pursuant to one or more, or a part of one or more, program modules or objects generated from any

programming language, such as C++, Java, Fortran or the like. And still further, each step, or a file or object or the like implementing each step, may be executed by special purpose hardware or a circuit module designed for that purpose.

In the case of diagrams depicted herein, they are provided by way of example. There may be variations to these diagrams or the steps (or operations) described herein without departing from the spirit of the invention. For instance, in certain cases, the steps may be performed in differing order, or steps may be added, deleted or modified. All of these variations are considered to comprise part of the present invention as recited in the appended claims.

Throughout the description and claims of this specification, the word "comprise" and variations of the word, such as "comprising" and "comprises", is not intended to exclude other additives, integers or processed steps.

While the preferred embodiment of this invention has been described in relation to the XML language, this invention need not be solely operate using the XML language. It will be apparent to those skilled in the art that the invention may equally be operable with other computer languages, such as SGML.

The invention is preferably implemented in a high level procedural or object-oriented programming language to

communicate with a computer. However, the invention can be implemented in assembly or machine language, if desired. In any case, the language may be a compiled or interpreted language.

5 While aspects of the invention relate to certain computer language and other technological specifications (e.g. the XML specification), it should be apparent that classes, objects, components, tags and other such software and technological items referenced herein need not fully
10 conform to the specification(s) defined therefor but rather may meet only some of the specification requirements. Moreover, the classes, objects, components, tags and other such software and technological items referenced herein may be defined according to equivalent specification(s) other
15 than as indicated herein that provides equivalent or similar functionality, constraints, etc. For example, instead of the XML language specification, tags and other such software and technological items referenced herein may be defined according to the SGML specification where applicable and
20 appropriate.

 The invention may be implemented as a mechanism or a computer program product comprising a recording medium. Such a mechanism or computer program product may include, but is not limited to, CD-ROMs, diskettes, tapes, hard drives,
25 computer RAM or ROM and/or the electronic, magnetic, optical, biological or other similar embodiment of the program. Indeed, the mechanism or computer program product

may include any solid or fluid transmission medium, magnetic or optical, or the like, for storing or transmitting signals readable by a machine for controlling the operation of a general or special purpose programmable computer according to the method of the invention and/or to structure its components in accordance with a system of the invention.

The invention may also be implemented in a system. A system may comprise a computer that includes a processor and a memory device and optionally, a storage device, an output device such as a video display and/or an input device such as a keyboard or computer mouse. Moreover, a system may comprise an interconnected network of computers. Computers may equally be in stand-alone form (such as the traditional desktop personal computer) or integrated into another apparatus (such a cellular telephone). The system may be specially constructed for the required purposes to perform, for example, the method steps of the invention or it may comprise one or more general purpose computers as selectively activated or reconfigured by a computer program in accordance with the teachings herein stored in the computer(s). The procedures presented herein are not inherently related to a particular computer system or other apparatus. The required structure for a variety of these systems will appear from the description given.

While this invention has been described in relation to preferred embodiments, it will be understood by those skilled in the art that changes in the details of

construction, arrangement of parts, compositions, processes, structures and materials selection may be made without departing from the spirit and scope of this invention. Many modifications and variations are possible in light of the
5 above teaching. Thus, it should be understood that the above described embodiments have been provided by way of example rather than as a limitation and that the specification and drawing(s) are, accordingly, to be regarded in an illustrative rather than a restrictive sense.

10 The present invention can be included in an article of manufacture (e.g., one or more computer program products) having, for instance, computer usable media. The media has embodied therein, for instance, computer readable program code means for providing and facilitating the capabilities
15 of the present invention. The article of manufacture can be included as a part of a computer system or sold separately.

20 Additionally, at least one program storage device readable by a machine, tangibly embodying at least one program of instructions executable by the machine to perform the capabilities of the present invention can be provided.

25 The flow diagrams depicted herein are just examples. There may be many variations to these diagrams or the steps (or operations) described therein without departing from the spirit of the invention. For instance, the steps may be performed in a differing order, or steps may be added, deleted or modified. All of these variations are considered a part of the claimed invention.

Although preferred embodiments have been depicted and described in detail herein, it will be apparent to those skilled in the relevant art that various modifications, additions, substitutions and the like can be made without departing from the spirit of the invention and these are therefore considered to be within the scope of the invention as defined in the following claims.

[illegible]

APPENDIX I

DTD Specifications

663030 "663030"

```
<?xml encoding="US-ASCII"?>
<!ENTITY % tdechildren "(#PCDATA | outfile | define | clear
5 | if | repeat |
scope | repeatscope | createalias |
include | code | call | sameline | assert | ifhasscope |
ifhasrepeatscope | hasscope | targetscope )" >
<!ENTITY % targettdechildren "(#PCDATA | outfile | define |
10 clear | if | repeat | scope | repeatscope | createalias |
include | code | call | sameline | assert | ifhasscope |
repeattargetscope |
ifhasrepeatscope | hasscope | updatetargetscope |
addtargetscope |
15 removetargetscope | ifhastargetscope |
ifhasrepeattargetscope | hastargetscope |
targetscope | updatetargetdoctype )" >
<!ENTITY % ifhastdechildren "(#PCDATA | outfile | define |
clear | if | else |
20 repeat | scope | repeatscope | createalias |
include | code | call | sameline | assert | ifhasscope |
repeattargetscope |
ifhasrepeatscope | hasscope | updatetargetscope |
addtargetscope |
25 removetargetscope | ifhastargetscope |
ifhasrepeattargetscope | hastargetscope |
targetscope | updatetargetdoctype )" >
```

```

<!ENTITY % elsetdechildren "(#PCDATA | outfile | define |
clear | if | repeat |
scope | repeatscope | createalias |
include | code | call | sameline | assert | ifhasscope |
5 ifhasrepeatscope |
hasscope | else | targetscope )*" >
<!ELEMENT define ( #PCDATA ) >
<!ATTLIST define MACRO CDATA #REQUIRED>
<!ELEMENT clear EMPTY >
10 <!ATTLIST clear MACRO CDATA #REQUIRED>
<!ELEMENT _TDEBlock_ %targettdechildren; >
<!ATTLIST _TDEBlock_ DOMTree CDATA #IMPLIED>
<!ELEMENT updatetargetdoctype EMPTY >
<!ATTLIST updatetargetdoctype SYSTEM_URL CDATA #IMPLIED>
15 <!ATTLIST updatetargetdoctype PUBLIC_URL CDATA #IMPLIED>
<!ATTLIST updatetargetdoctype PUBLIC_NAME CDATA #IMPLIED>
<!ATTLIST updatetargetdoctype ROOT_ELEMENT_NAME CDATA
#IMPLIED>
<!ELEMENT if %elsetdechildren; >
20 <!ATTLIST if EXPRESSION CDATA #REQUIRED>
<!ELEMENT else EMPTY >
<!ELEMENT addtargetscope %targettdechildren; >
<!ATTLIST addtargetscope NAME CDATA #REQUIRED>
<!ATTLIST addtargetscope TYPE (ELEMENT | ATTRIBUTE)
25 "ELEMENT">
<!ATTLIST addtargetscope INDEX CDATA #IMPLIED >
<!ELEMENT updatetargetscope %targettdechildren; >
<!ATTLIST updatetargetscope NAME CDATA #REQUIRED>
<!ATTLIST updatetargetscope TYPE (ELEMENT | ATTRIBUTE)
30 "ELEMENT">

```



```

<!ELEMENT code %tdechildren; >
<!ATTLIST code NAME CDATA #REQUIRED>
<!ELEMENT call %tdechildren; >
<!ATTLIST call NAME CDATA #REQUIRED>
5 <!ATTLIST call MACROLIST CDATA #IMPLIED>
<!ELEMENT sameline %tdechildren; >
<!ELEMENT assert EMPTY >
<!ATTLIST assert EXPRESSION CDATA #REQUIRED>
<!ELEMENT ifhasrepeatscope %ifhastdechildren; >
10 <!ATTLIST ifhasrepeatscope NAME CDATA #REQUIRED>
<!ELEMENT hasscope %tdechildren; >
<!ATTLIST hasscope NAME CDATA #REQUIRED>
<!ELEMENT ifhasscope %ifhastdechildren; >
<!ATTLIST ifhasscope NAME CDATA #REQUIRED>
15 <!ELEMENT createalias %targettdechildren; >
<!ATTLIST createalias ALIASNAME CDATA #REQUIRED>
<!ATTLIST createalias ALIASPATH CDATA #REQUIRED>

```

APPENDIX II

Background

5 The preferred embodiment of the mechanism according to
the present invention for updating a domain model and
generating a formatted output works with an XML model,
however, the mechanism is unaware of the structure of the
model. It only responds to navigational, substitution and
emit directives contained in the templates. The model is
traversed in a symbolic fashion (i.e. naming the object
relationships as strings, rather than direct object
10 references), and the attribute values are retrieved in the
same fashion (i.e. by naming the attribute in a string).

Language Specification

Introduction

15 For ease of use, the template language is a simple language
that has a well formed XML syntax. The language consists of
the following constructs:

20 1. Text. Text is emitted as is. Blanks are treated as text;
any indentation that occurs in the template file will be
echoed in the output.

2. Macros. Macros associate a name with a set of values.
They enable string substitution within templates.

3. Directives. Directives are commands that control code
generation.

4. Comments. Comments help to clarify the template and do not get emitted into the output file.

Throughout this Appendix there are references to the '[' and ']' character. These characters emphasize that the

5 parameter within these characters are optional. For example consider the following:

transform = toupper [index]

wherein the 'index' parameter is optional.

Template Files

10 Template files are flat text files that are portable across different platforms. They can be edited with any text editor. White space is treated as-is and new line characters are expected at the end of each line. In the preferred embodiment, the template files conform to the XML
15 specifications on each platform.

Macros

Macros enable string substitution within templates.

Macros Syntax

\$macro_name\$

20 Description

\$macro_name\$ defines a macro name that will be substituted with its assigned string value. The macro instance is replaced with the macro value as the output is emitted. A

macro name consists of lower and upper case letters, and digits.

A macro is an implicit array: a simple macro has either one string value, or a (ordered) sequence of string values; a
5 macro structure is either a structure of macro fields (which, in turn, are implicit arrays) or it is an array of identical structures. Here identical structures is meant to imply that all the fields of the structures have the same names.

10 Example 1 (simple macro)
Consider the following template definition:

..
class \$CLASS_NAME\$
{
15 };
..

If the user defines the macro CLASS_NAME to be GuiObject, then the following will be emitted:

20 class GuiObject
{
};

Macro Transformations

Macro transformations allow you to take a macro value, convert it to some other form, and then use the result of the transformation as the macro value. A transformation can
5 be used whenever a macro is used.

Syntax

\$macro_name| transform\$

The first transformation type (|) does not permanently change the value of the macro, but only during that
10 particular macro emit. The second transformation type (=) permanently changes the value of the macro.

The mechanism recognizes the following transformations:
(note: [parameter] means optional parameter and the user must provide an index when using "[]")

15 transform = toupper [i] |
 tolower [i] |
 truncate [maxlen] |
 separate separator |
20 strip [pattern] |
 stripTrailing [pattern] |
 stripLeading [pattern] |
 change in_pattern out_pattern |
 increment [i] |
25 numWords |
 words [i] |
 word[i]

Description

The following describes each transform:

5 toupper - converts the macro value to all upper case
 letters. If the optional parameter i is specified, then
 only the i'th character is converted to upper case.

 tolower - converts the macro value to all lower case
 letters. If the optional parameter i is specified, then
 only the i'th character is converted to upper case.

10 truncate - truncates the macro value to maxlen
 characters long by removing vowels. With no parameter,
 the default is to truncate to 8 characters.

15 separate - separates the values in a multivalue macro
 by a separator. If the separator is a string/character,
 it must be enclosed by quotes ''. Otherwise it is
 assumed to be a macro name and the value of that macro
 is used as a separator. When used in a <repeat>
 statement, repeat blocks are separated by a separator.

20 strip - removes all leading and trailing white space
 characters from the macro value. If a string pattern is
 specified (either as a string or a macro name), the
 pattern will be stripped off from the macro value.

 stripTrailing - removes all trailing white space
 characters from the macro value. An optional pattern to
 strip can be specified

1. The first of these is the fact that the
 2.
 3.
 4.
 5.
 6.
 7.
 8.
 9.
 10.
 11.
 12.
 13.
 14.
 15.
 16.
 17.
 18.
 19.
 20.
 21.
 22.
 23.
 24.
 25.
 26.
 27.
 28.
 29.
 30.
 31.
 32.
 33.
 34.
 35.
 36.
 37.
 38.
 39.
 40.
 41.
 42.
 43.
 44.
 45.
 46.
 47.
 48.
 49.
 50.
 51.
 52.
 53.
 54.
 55.
 56.
 57.
 58.
 59.
 60.
 61.
 62.
 63.
 64.
 65.
 66.
 67.
 68.
 69.
 70.
 71.
 72.
 73.
 74.
 75.
 76.
 77.
 78.
 79.
 80.
 81.
 82.
 83.
 84.
 85.
 86.
 87.
 88.
 89.
 90.
 91.
 92.
 93.
 94.
 95.
 96.
 97.
 98.
 99.
 100.
 101.
 102.
 103.
 104.
 105.
 106.
 107.
 108.
 109.
 110.
 111.
 112.
 113.
 114.
 115.
 116.
 117.
 118.
 119.
 120.
 121.
 122.
 123.
 124.
 125.
 126.
 127.
 128.
 129.
 130.
 131.
 132.
 133.
 134.
 135.
 136.
 137.
 138.
 139.
 140.
 141.
 142.
 143.
 144.
 145.
 146.
 147.
 148.
 149.
 150.
 151.
 152.
 153.
 154.
 155.
 156.
 157.
 158.
 159.
 160.
 161.
 162.
 163.
 164.
 165.
 166.
 167.
 168.
 169.
 170.
 171.
 172.
 173.
 174.
 175.
 176.
 177.
 178.
 179.
 180.
 181.
 182.
 183.
 184.
 185.
 186.
 187.
 188.
 189.
 190.
 191.
 192.
 193.
 194.
 195.
 196.
 197.
 198.
 199.
 200.
 201.
 202.
 203.
 204.
 205.
 206.
 207.
 208.
 209.
 210.
 211.
 212.
 213.
 214.
 215.
 216.
 217.
 218.
 219.
 220.
 221.
 222.
 223.
 224.
 225.
 226.
 227.
 228.
 229.
 230.
 231.
 232.
 233.
 234.
 235.
 236.
 237.
 238.
 239.
 240.
 241.
 242.
 243.
 244.
 245.
 246.
 247.
 248.
 249.
 250.
 251.
 252.
 253.
 254.
 255.
 256.
 257.
 258.
 259.
 260.
 261.
 262.
 263.
 264.
 265.
 266.
 267.
 268.
 269.
 270.
 271.
 272.
 273.
 274.
 275.
 276.
 277.
 278.
 279.
 280.
 281.
 282.
 283.
 284.
 285.
 286.
 287.
 288.
 289.
 290.
 291.
 292.
 293.
 294.
 295.
 296.
 297.
 298.
 299.
 300.
 301.
 302.
 303.
 304.
 305.
 306.
 307.
 308.
 309.
 310.
 311.
 312.
 313.
 314.
 315.
 316.
 317.
 318.
 319.
 320.
 321.
 322.
 323.
 324.
 325.
 326.
 327.
 328.
 329.
 330.
 331.
 332.
 333.
 334.
 335.
 336.
 337.
 338.
 339.
 340.
 341.
 342.
 343.
 344.
 345.
 346.
 347.
 348.
 349.
 350.
 351.
 352.
 353.
 354.
 355.
 356.
 357.
 358.
 359.
 360.
 361.
 362.
 363.
 364.
 365.
 366.
 367.
 368.
 369.
 370.
 371.
 372.
 373.
 374.
 375.
 376.
 377.
 378.
 379.
 380.
 381.
 382.
 383.
 384.
 385.
 386.
 387.
 388.
 389.
 390.
 391.
 392.
 393.
 394.
 395.
 396.
 397.
 398.
 399.
 400.
 401.
 402.
 403.
 404.
 405.
 406.
 407.
 408.
 409.
 410.
 411.
 412.
 413.
 414.
 415.
 416.
 417.
 418.
 419.
 420.
 421.
 422.
 423.
 424.
 425.
 426.
 427.
 428.
 429.
 430.
 431.
 432.
 433.
 434.
 435.
 436.
 437.
 438.
 439.
 440.
 441.
 442.
 443.
 444.
 445.
 446.
 447.
 448.
 449.
 450.
 451.
 452.
 453.
 454.
 455.
 456.
 457.
 458.
 459.
 460.
 461.
 462.
 463.
 464.
 465.
 466.
 467.
 468.
 469.
 470.
 471.
 472.
 473.
 474.
 475.
 476.
 477.
 478.
 479.
 480.
 481.
 482.
 483.
 484.
 485.
 486.
 487.
 488.
 489.
 490.
 491.
 492.
 493.
 494.
 495.
 496.
 497.
 498.
 499.
 500.
 501.
 502.
 503.
 504.
 505.
 506.
 507.
 508.
 509.
 510.
 511.
 512.
 513.
 514.
 515.
 516.
 517.
 518.
 519.
 520.
 521.
 522.
 523.
 524.
 525.
 526.
 527.
 528.
 529.
 530.
 531.
 532.
 533.
 534.
 535.
 536.
 537.
 538.
 539.
 540.
 541.
 542.
 543.
 544.
 545.
 546.
 547.
 548.
 549.
 550.
 551.
 552.
 553.
 554.
 555.
 556.
 557.
 558.
 559.
 560.
 561.
 562.
 563.
 564.
 565.
 566.
 567.
 568.
 569.
 570.
 571.
 572.
 573.
 574.
 575.
 576.
 577.
 578.
 579.
 580.
 581.
 582.
 583.
 584.
 585.
 586.
 587.
 588.
 589.
 590.
 591.
 592.
 593.
 594.
 595.
 596.
 597.
 598.
 599.

5

10

15

15

20

Consider the following template definition:

CA919990037US1

```

/*
 * File name: $CLASS_NAME|truncate$
 */

```

5 If the user defines the macro CLASS_NAME to be GuiObject,
then the following will be emitted:

```

#define _GUIOBJECT_HPP_
/*
 * File name: GiObject
10 */

```

Example 2:

Consider the following template:

```

<_TDEBlock_>
<define MACRO="FILE_NAME">myFile.hpp</define>
15 <define MACRO="EXTENSION">hpp</define>
    $FILE_NAME| change EXTENSION ``cpp``$
</_TDEBlock_>

```

20 The above template file defines a macro called FILE_NAME
whose string value is myFile.hpp and EXTENSION is a macro
whose string value is hpp. For information regarding the
define directive, refer below. The resulting output is
emitted:

myFile.cpp

Example 3:

25 Consider the following template:

CA919990037US1

```

<_TDEBlock_>
<define MACRO="firstname">John</define>
<define MACRO="lastname">Doe</define>
  $firstname= toupper$ $lastname= toupper$
5   $firstname$ $lastname$
</_TDEBlock_>

```

In this case, the "=" transformation type is used instead of the "|" transformation type. Note that this template file defines a firstname and lastname macro. The values of the

10 firstname and the lastname macros are uppercased. Once this is done the values of the firstname and lastname macros are always uppercased. For information regarding the define directive, refer below. The resulting output is emitted:

```

JOHN DOE
15 JOHN DOE

```

Note that the values of firstname and lastname macros retain their transformation (ie the value of each macro is uppercased).

Directives

20 Directives define the flow of control in the template files. Directives are enclosed within '< >' delimiters (e.g. <name>) and can appear anywhere in the template. XML imposes some restrictions on attribute values. For example to form a well-formed XML document the characters '<' and

25 '&' may only be used to start tags and entities. Therefore,

entity references must be used to represent the '<' and '&' characters. The following is list of entity references:

Entity Reference Character Representation

	&	&
5	<	<
	>	>
	'	'
	&qout;	"

10 Certain directives must be used in begin/end pairs using the notation <name> ... </name>.

A directive with nothing inside is an empty directive and has no effect in the emitted text. If a directive starts in the middle of a line, the leading white space is taken as indentation for the language and it will not be emitted.

15 The following directives are supported:

	TDEBlock
	define
	clear
	if
20	repeat
	scope
	repeatscope
	include
	outfile
25	code
	call

- sameline
- assert
- hasscope
- ifhasscope
- 5 ifhasrepeatscope
- targetscope
- repeattargetscope
- ifhasrepeattargetscope
- addtargetscope
- 10 removetargetscope
- updatetargetscope
- updatetargetdoctype
- addattribute
- updateattribute
- 15 removeattribute
- addtext
- updatetext
- removetext
- hastargetscope
- 20 ifhastargetscope
- createalias

`_TDEBlock_ Directive`

`_TDEBlock_` is the only required directive that should be the root element for each template file.

- 25 Syntax
- `<_TDEBlock_ SourceDOMTree="sourcedoc.xml"`
- `TargetDOMTree="targetdoc.xml">`

section

</_TDEBlock_>

Description

5 A source and target XML document can be specified in this directive. By specifying a target XML document one can write directives to modify the targeted XML document. By specifying a source XML document one can read data from the XML document. For more information see the "Modifying Existing DOM Trees" section.

10 Example 1: Simple template file

<_TDEBlock_>

</_TDEBlock_>

Example 2: Simple template file that specifies a target XML document

15 <_TDEBlock_ TargetDOMTree="doc.xml">

</_TDEBlock_>

Example 3: Simple template file that specifies a target XML document and a source XML document.

<_TDEBlock_ SourceDOMTree="sourcedoc.xml"

20 TargetDOMTree="doc.xml">

</_TDEBlock_>

define Directive

The define directive associates the specified value, macro_value, with the macro_name. NOTE: <define> can only be used to define simple macros. \

Syntax

5 <define MACRO="macro_name [|
transformation]">value</define>

Description

The define directive takes a single macro_value and assigns it to macro_name. The macro_value will be interpreted as is.
10 If a transformation is present, macro_value is transformed before being assigned to macro_name. Multiple values can be assigned to macro_name by performing subsequent definitions. For macros that have multiple value assignments, the values are concatenated together (without separators) and treated
15 as a single string. See below for multi-valued macros usage in a repeat directive. The macro definition can be removed by the clear directive.

NOTE:

20 <define MACRO="MACRO">value</define>
is different from
 <define MACRO="MACRO">
 value
 </define>
as an extra newline character is added by the later.

25 Example 1: Single-value macro
Consider the following template definition:
 <_TDEBlock_>

<define MACRO="String">

Hi there. \

How are you?

</define>

5 \$String\$

</_TDEBlock_>

The resulting output emitted is:

Hi there. How are you?

10 The backslash at the end of the first line prevents the end of line character from being embedded in the string. If '\\' is omitted, the text will be emitted on two lines as is.

Example 2: Multi-value macro

Consider the following template definition:

15 <_TDEBlock_>

<define MACRO="T_Class">Parent_</define>

<define MACRO="T_Class">Class</define>

<define MACRO="M_Class">my_class</define>

class \$M_Class\$: public \$T_Class\$

20 {

ARG

};

</_TDEBlock_>

When you run the TDE, the following is emitted:

25

class my_class : public Parent_Class

CA919990037US1

```
{
  ARG
};
```

Example 3: Transformation applied to macros

5 Consider the following template definition:

```
<?xml version="1.0"?>
<!DOCTYPE _TDEBlock_ SYSTEM "../.../tde.dtd">
<_TDEBlock_>
  <outfile FILENAME="define.log">
10    <define MACRO="macro1|toupper|strip E">one</define>
    <define MACRO="macro2|tolower|strip E">one</define>
    <define MACRO="macro3|tolower|strip e">ONE</define>
    $macro1$
    $macro2$
15    $macro3$
    </outfile>
  <_TDEBlock_>
```

The following is emitted to the "define.log" file:

```
20    ON
    one
    on
```

25 The above template file defines a macro called "macro1" that goes through two transforms. The first transform uppercases the value of this macro, in this case the value changes from "one" to "ONE". After that transformation, the "E" is stripped from the end. The resulting macro is "ON". The

issuing a clear on a macro which has not been defined has no effect.

Example

The following clears the macro MEMBER_NAME.

```
5  <_TDEBlock_>
    <define MACRO="MEMBER_NAME">foo</define>
    <clear MACRO="MEMBER_NAME">
    $MEMBER_NAME$
    </_TDEBlock_>
```

```
10 The resulting emitted output is as follows:
    $MEMBER_NAME$
```

The macro MEMBER_NAME has a value of "foo" then it is cleared. When the macro is cleared it is as though the macro was never defined.

```
15 Therefore the generated output would print out the macro
    name instead of the value (i.e. $MEMBER_NAME$).
```

if Directive

```
20 The if directive facilitates the emission of a section of
    code based on some condition(s). The syntax used for
    filtering is similar to that used for conditional branches
    in C.
```

Syntax

```
<if EXPRESSION="expr">
<else/>
25 </if>
```

where,

```
expr= (expr logical-operator expr)
      (value cond value)
      ($macro_name$)
5      (!expr)
      (!$macro_name$)
      logical-operator = & | |
      cond = == | != | < | > | <= | >=
      value = string | $macro_name$
```

10 Description

The conditional expression must be fully parenthesized. An expression takes the form of a value, followed by a condition, then followed by a value. The value can be a string or another macro_name. The expression evaluates to a string comparison. The else clause is optional. If the expression is too long, you can use the ''\' as an escape character to specify it on multiple lines. The ! is used as a negation operator. If strings contain blanks they can be enclosed by quotes. The '<' and '>' operators are used for string containment: A < B evaluates to true if A is a substring of B (e.g. S1 < S2 evaluates if S1 is a substring of S2.) On the other hand, '<=' and '>=' have a different interpretation compared to '<' and '>'. '<=' and '>=' are relational operators used for comparing expressions. (e.g

15

20

25 (value <=1) evaluates if value is smaller or equal to 1.)

Example #1

This example illustrates the use of nested ifs:

<_TDEBlock_>

CA919990037US1

```

    <if EXPRESSION="($METHOD_NAME$ == &qout;queue&qout;)">
    code to emit if method name is ``queue''
    <else/><if EXPRESSION="($METHOD_NAME$ ==
&qout;print&qout;)">
5      code to emit if method name is ``print''
    <else/>
      code to emit otherwise...
    </if></if>
</_TDEBlock_>

```

10 Example #2

This example illustrates the use of the less-than (<) operation and the and (&) operation. The conditional operators act on string values.

```

<_TDEBlock_>
15 <outfile FILENAME="if.log">
  <define MACRO="TotalMethods"> print delete add </define>
  <define MACRO="MY_METHOD">print</define>
  <define MACRO="DONE">no</define>
  <if EXPRESSION="( (!($MY_METHOD$ &lt; $TotalMethods$)) &amp;
20 ( $DONE$ != &qout;yes&qout;))">
    code to emit if we're not done and my method is
    unknown
  </if>
  </outfile>
25 </_TDEBlock_>

```

The emitted code is as follows:

The repeat directive will emit a section, within the start and end delimiters, repeatedly for each value associated with macro_name. The macro-list defines which macros to iterate over. If a macro has multiple values, each value
5 will be emitted once in the repeat section. In essence, the repeat is analogous to a for loop, looping through all the macro values. If a transformation is present, it applies to the text contained within the repeat section. The number of iterations is the smallest number of values the macros have.
10 If one of the macros is undefined, the code inside the <repeat> ... </repeat> will not be emitted.

Example 1

Consider the following template definition:

```
<_TDEBlock_>  
15     <define MACRO="COUNT">1</define>  
        <define MACRO="COUNT">2</define>  
        <define MACRO="COUNT">3</define>  
        <repeat MACRO="COUNT">  
            list.add(person$COUNT$);  
20     </repeat>  
    </_TDEBlock_>
```

When the template is parsed, the following is emitted:

```
        list.add(person1);  
        list.add(person2);  
25     list.add(person3);
```

Example 2

Consider the following template definition:

CA919990037US1


```

void func(
    int i,
    char c
);

```

5 scope Directive

The scope directive is used to navigate the XML model and specify how the macros in a section are scoped.

Syntax

```

10 <scope NAME="sname">
    section
</scope>

```

Description

The sname specifies the name of the scope.

Example

15 Suppose the model defines an element ''Class'' which has two other embedded elements.

```

    <Class name="Set">
    <Method name="add">
20 </Method>
    </Class>

```

When the element ''Class'' is generated, using the following template:

```

25 <_TDEBlock_>

```

```
class $name$ {          <-- this $name$ is from the Class
object
```

```
    <scope NAME="Method">
        void $name$();
5    </scope>
};
</_TDEBlock_>
```

this will be emitted as:

```
10 class Set {
    void add();
}
```

In order to make scoping conceptually easy to use, a
'directory' navigation approach has been adopted. By
15 specifying a '//' an implicit scope operation is implied.
For example, consider the following XML document.

```
<classlist>
    <class classname="MyClass">
        <datamember>Member1</datamember>
20    </class>
</classlist>
```

And the following template file:

```
<_TDEBlock_>
25    <scope NAME="class">
        <scope NAME="datamember">
```

```

        </scope>
    </scope>
</_TDEBlock_>

```

The above template file can also be written as:

5

```

<_TDEBlock_>
<scope NAME="class//datamember">
</scope>
</_TDEBlock_>

```

10

It is also possible to write a template without using the scope directive. Consider the following XML document.

```

<Class name="Set">
<Method name="add">
</Method>
15 </Class>

```

Also consider the following template file.

20

```

<_TDEBlock_>
class $name$ {          <-- this $name$ is from the Class
object
    void $Method//name$();
};
</_TDEBlock_>

```

The following is the resulting emitted code:

```

class Set {

```

```

        void add();
    }

```

The scoping navigation starts at the current scope. However, navigation can occur relative to the root element by specifying a '/' in front of the scoping pattern. Parent navigation can also be achieved by specifying '..' within the scoping pattern. Consider the following XML document.

```

10  <classlist>
    <class classname="MyClass">
        <datamember>Member1</datamember>
    </class>
</classlist>

```

And, the following template file.

```

15  <_TDEBlock_>
    <scope NAME="class">
        <scope NAME="datamember">
            <scope NAME="..">
20      $classname$
            </scope>
        </scope>
    </scope>
</_TDEBlock_>

```

25 The following would be emitted:

```

    MyClass

```

An XML document can consists of elements of the same type. In order to navigate elements of the same type an index is specified to represent a specific element. For example the following XML document consists of three elements of the same type.

```
<classlist>
  <class>
    <method>init</method>
    <method>read</method>
    <method>write</method>
  </class>
</classlist>
```

Scoping can occur based on the position of the element. This is illustrated in the following template file.

```
<_TDEBlock_>
<scope NAME="class">
  <scope NAME="method[1]">
    $TEXT$
  </scope>
  <scope NAME="method[2]">
    $TEXT$
  </scope>
  <scope NAME="method[0]">
    $TEXT$
  </scope>
</scope>
```

</_TDEBlock_>

The following would be emitted:

```
5      read
      write
      init
```

repeatscope Directive

10 The repeatscope directive allows a single section of a
template to emit over and over again on a list of elements.
It acts as an element iterator.

Syntax

```
15 <repeatscope NAME="sname" SEPARATOR="separator">
    section
    </repeatscope>
```

Description

The sname specifies the name of the scope. The separator specifies the delimiter to use between sections. The default value is an empty string.

20 Example 1

Suppose the model defines an element 'Class' which has two other embedded elements.

```
25 <Classlist>
    <Class name="Set">
        <Method name="add">
        </Method>
```

CA919990037US1


```

<Method name="del">
</Method>
</Class>
</Classlist>

```

- 5 When the element 'Class' is generated, using the following template:

```

<_TDEBlock_>
<scope NAME="Class">
10   <repeatscope NAME="Method">
        $name$
    </repeatscope>
</scope>
</_TDEBlock_>

```

- 15 The following would be emitted.
- ```

 add
 del

```

## Example 2

- 20 Consider the following XML document. This example shows how one can use the 'SEPARATOR' attribute.

```

<Elementlist>
<Element name="A"/>
<Element name="B"/>
<Element name="C"/>
25 <Element name="D"/>
<Element name="E"/>

```

```

<Element name="F"/>
<Element name="G"/>
</Elementlist>

```

And, the following template file.

```

5 <_TDEBlock_>
 <repeatscope NAME="Element" SEPARATOR=", ">
$name$
 </repeatscope>
</_TDEBlock_>

```

10 The resulting emitted code is as follows:  
A,B,C,D,E,F,G

include directive

15 The include directive imbeds a single input template file  
in-line.

Syntax

```
<include FILENAME="input_file_name"/>
```

Description

20 The include directive includes an entire file specified by  
input\_file\_name, into the input stream for processing. It is  
analogous to the #include file\_name C directive. The  
input\_file\_name must be an input template file.

Example

25 A file called comment.tde contains the following:

CA919990037US1

```

//-----
// C++ Generated Code
//-----

```

Consider the following template file which includes  
 5 comment.tde:

```

<_TDEBlock_>
<include FILENAME="comment.tde"/>
 class A {};
10 </_TDEBlock_>

```

The following is emitted:

```

//-----
// C++ Generated Code
15 //-----
 class A {};

```

outfile Directive

The outfile directive specifies the output file to write to  
 for that section of the template.

20 Syntax

```

<outfile FILENAME="outfile_name" MODE="_mode_">
section
</outfile>

```

where,

```

25 outfile_name = $macro_name$ | string
 mode = append | new

```

CA919990037US1

## Description

When the outfile directive is encountered, the specified outfile\_name is opened with the specified mode, and the section of the template is written to that output file.

### 5 Example

Consider the following template definition:

```
60663" 637363
<_TDEBlock_>
<outfile FILENAME="$filename$" MODE="new">
10 // C++ generated code
foo();
<outfile FILENAME="d:\\log.dat" MODE="append"> <-- need
to escape the backslash
This is a log entry.
15 </outfile>
// End C++ generated code
</outfile>
</_TDEBlock_>
```

20 If the macro filename is defined to be foo.cpp, the following is emitted to foo.cpp:

```
// C++ generated code
foo();
// End C++ generated code
25 The file d:\log.dat will contain the following:
```

This is a log entry.

## code Directive

The code directive identifies a section of code to emit when the cname value is set.

### Syntax

```
5 <code NAME="cname">
 section
</code>
```

A code section can also be declared with parameters, in which case the call directive must be used to emit it.

```
10 <code NAME ="cname, param1, param2, ..., param_n">
 section
</code>
```

### Description

```
15 The code directive takes a single cname and emits the
delimited section if cname is invoked. If cname is not set,
the section is not emitted. Code sections are useful for
reusing templates, or for overriding other code sections
with the same name. When using the <scope> directive, code
20 sections can also be invoked recursively. The terminating
condition is implicitly defined by the model, that is, when
the code section is invoked from within a <scope> section
that will eventually not be reached.
```

### Example1

```
25 Consider the following template:
```

CA919990037US1

```

 <_TDEBlock_>
 <code NAME ="C_FORM">
 malloc(sizeof(Fred));
5 </code>
 <code NAME ="CPP_FORM">
 Fred* pTemp = new Fred;
 </code>
 <call NAME ="CPP_FORM"/>
10 return ok;
</_TDEBlock_>

```

The following is emitted:

```

 Fred* pTemp = new Fred;
15 return ok;

```

call directive

The call directive is the only way of invoking code sections. (Recall that a code section can also be emitted as a macro, by enclosing its name with \$. This only works for code sections with no parameters).

Syntax:

```

<call NAME="codeName" MACROLIST="macro1 macro2 ...
macro_n"/>

```

All the arguments of the call directive (a code name followed by macro names) can be separated by commas.

Example:

CA919990037US1

```

<_TDEBlock_>
<code NAME="C, M, N">
 M = M
 <repeat MACRO="N">
5 N = N
 </repeat>
</code>
<call NAME="C" MACROLIST="M, N"/>
</_TDEBlock_>

```

10 If macro M was defined to have value 'foo' and N to have two values, 'bar' and 'BAR' then the following is emitted:

```

 M = foo
15 N = bar
 N = BAR

```

sameline Directive

Syntax

```

<sameline>
20 section
</sameline>

```

Description

The sameline directive is used to put its contained section on a single output line. Basically, all the newline characters from the section are removed and what follows

25 after </sameline> will start on a new line.

CA919990037US1

Example1

Consider the following template:

```

5 <_TDEBlock_>
 <sameline>
 int func(
 <repeat MACRO="PARMS | separate ',' '>
 $PARMS$
 </repeat>
10);
 </sameline>
</_TDEBlock_>
```

When the template is parsed with PARMS containing values  
"int a" and "char c", the  
15 following is emitted:

```
 int func(int a, char c);
```

assert Directive

Syntax

```

20 <assert EXPRESSION="expr"/>
 expr= (expr logical-operator expr)
 (value cond value)
 ($macro_name$)
 (!expr)
25 (!$macro_name$)
 logical-operator = & | |
```

CA919990037US1



cond = == | != | < | > | <= | >=  
value = string

#### Description

The assertion directive is used to test a boolean expression. If this expression is evaluated to false, an assertion failure exception is thrown.

#### Example1

Consider the following template:

```
<_TDEBlock_>
10 <define MACRO="macroA">Bird</define>
 <define MACRO="macroB">Bee</define>
 <sameline>
 int func(
15 <assert EXPRESSION="$macroA$ < $macroB$"/>
);
 </sameline>
</_TDEBlock_>
```

When the template is parsed nothing is emitted since Bird is not a substring of Bee. Instead an assertion failure exception is thrown.

#### hasscope directive

The hasscope directive is used to determine if a scope exists within the XML model.

#### 25 Syntax

```
<hasscope NAME="sname">
```

CA919990037US1

section  
</hasscope>

#### Description

The sname specifies the name of the scope. This directive  
5 does not navigate the XML model. If the scope name does  
exist the 'section' is emitted.

#### Example

Suppose the model defines an element ''Class'' which has two  
other embedded elements:

10 <Class name="Set">  
<Method name="add">  
</Method>  
<Method name="del">  
</Method>  
15 </Class>

When the part ''Class'' is generated, using the following  
template:

<\_TDEBlock\_>  
20 class \$name\$ { <-- this \$name\$ is from the Class  
object  
    <hasscope NAME="Method">  
        /\*Starting Method Declaration\*/  
        <scope NAME="Method">  
25           void \$name\$();  
        </scope>  
        /\*End of Method Declaration\*/

```

 </hasscope>
};
</_TDEBlock_>

```

The following is emitted:

5

```

class Set {
 /*Starting Method Declaration*/
 void add();
 void del();
10 /*End of Method Declaration*/
}

```

ifhasscope directive

15

The ifhasscope directive behaves in the same manner as a hasscope directive followed by a scope directive. This directive first tests whether the scope name exists. If the scope name does exist the scope name is navigated. This directive may be used in conjunction with an else statement.

Syntax

20

```

<ifhasscope NAME="sname">
sectionA
<else/>
sectionB
</ifhasscope>

```

25

Description

The sname specifies the name of the scope

CA919990037US1

Example

Suppose the model defines an element ``Class'' which has two other embedded elements:

```
5 <Class name="Set">
 <Method name="add">
 </Method>
 <Method name="del">
 </Method>
10 </Class>
```

When the part ``Class'' is generated, using the following template:

```
<_TDEBlock>
15 class $name$ {
 <ifhasscope NAME="Method">
 /*Starting Method Declaration*/
 void $name$();
 /*End of Method Declaration*/
20 <else/>
 method was not there
 </ifhasscope>
 };
</_TDEBlock_>
```

25 The following is emitted:

```
class Set {
```

CA919990037US1

```

 /*Starting Method Declaration*/
 void add();
 /*End of Method Declaration*/
 }

```

- 5 Note only the first method is emitted. In order to emit the second method a second "ifhasscope" directive must be added to the template file. If it were not possible to scope to method, the else statement would have been executed.

ifhasrepeatscope directive

- 10 The ifhasrepeatscope directive behaves in the same manner as a hasscope directive followed by a repeatscope directive. This directive first tests whether the scope name exist. If the scope name does exist all children elements that matches the scope name are navigated. This directive works in conjunction with the else statement.
- 15

Syntax

- ```

<ifhasrepeatscope NAME="sname">
    sectionA
<else/>
20 sectionB
</ifhasrepeatscope>

```

Description

The sname specifies the name of the scope

Example

CA919990037US1

Suppose the model defines an element ``Class'' which has two other embedded elements:

```
5      <Class name="Set">
        <Method name="add">
          </Method>
        <Method name="del">
          </Method>
      </Class>
```

10 When the part ``Class'' is generated, using the following template:

```
15  <_TDEBlock>
    class $name$ {
      <ifhasrepeatscope NAME="Method">
        /*Starting Method Declaration*/
        void $name$();
        /*End of Method Declaration*/
      <else/>
20      no methods were found
    </ifhasrepeatscope>
  };
</_TDEBlock_>
```

The following is emitted:

```
25  class Set {
      /*Starting Method Declaration*/
      void add();
```


Modifying Existing DOM Trees

Template files can be created to modify existing DOM trees associated with a domain model. There are several main operations that are defined to accomplish this task. Add an element, remove an element, scope to an element, update an attribute, update an element, add an attribute and remove an attribute.

targetscope directive

The targetscope directive navigates an element that is part of an existing DOM tree associated with a domain model.

Syntax

```
<targetscope NAME="sname" INDEX=_index_>
```

section

```
</targetscope>
```

where,

sname = target scope name

index = integer value

Description

The sname specifies the name of the scope. The _index_ value specifies which element to navigate based on an index value. The default value is set to 0. Therefore, if there is more than one element that is found, the first element is scoped.

Example

Suppose the model defines an element 'Class' which has two other embedded elements. This document is called class.xml.


```

<Classlist>
<Class name="Set">
<Method name="add">
</Method>
5  <Method name="del">
</Method>
</Class>
</Classlist>

```

10 Also consider the following template:

```

<?xml version="1.0"?>
<!DOCTYPE _TDEBlock_ SYSTEM "dtd/tde.dtd">
<_TDEBlock_ DOMTree="class.xml">
    <targetscope NAME="Class">
15    </targetscope>
</_TDEBlock_>

```

20 The mechanism will navigate to the Class element in the class.xml document. Note no emitted output is created since this directive just navigates the targeted XML document. In this case the targeted XML document is class.xml.

addtargetscope directive

The addtargetscope directive is used to insert an element into a DOM tree associated with a domain model.

Syntax

```

25 <addtargetscope NAME="sname" INDEX=_index_>
where,

```

CA919990037US1

sname = target scope name
index = integer value

Description

The sname specifies the name of the scope. The _index_ value specifies where to insert the element. The default value of the index is set to 0. Therefore, the element that is added will become the first child element. If a -1 is specified as the index the element that is added will be the last child element.

10 Example

Suppose the model defines an element ``Class'' which has two other embedded elements. Lets say that this document is called class.xml.

```
15 <Classlist>
    <Class name="Set">
      <Method name="add">
        </Method>
      <Method name="del">
        </Method>
20 </Class>
    </Classlist>
```

Also consider the following template:

```
25 <?xml version="1.0"?>
    <!DOCTYPE _TDEBlock_ SYSTEM "dtd/tde.dtd">
    <_TDEBlock_ DOMTree="class.xml">
```

```

<targetscope NAME="Class">
  <addtargetscope NAME="Method">
    <addtargetscope NAME="Argument"></addtargetscope>
  </addtargetscope>
5 </targetscope>
</_TDEBlock_>

```

The result of applying the template is as follows:

```

<ClassList>
<Class name="Set">
10 <Method>
    <Argument/>
  </Method>
  <Method name="add">
  </Method>
15 <Method name="del">
  </Method>
</Class>
</Classlist>

```

updatetargetscope directive

20 The updatetargetscope directive is used to update an element of a DOM tree associated with a domain model. If the element does not exist it is inserted into the DOM tree.

Syntax

```

<updatetargetscope NAME="sname" INDEX=_index_>
25 where,
    sname = target scope name
    _index_ = integer value

```

CA919990037US1

Description

The sname specifies the name of the scope. The default value of the index is set to 0. Therefore, the first element will be updated. The index value is only used when adding an element.

Example: Simple target document update

Suppose the model defines an element ''Class'' which has two other embedded elements. This document is called class.xml.

```
<Class name="Set">  
  <Method name="add">  
  </Method>  
  <Method name="del">  
  </Method>  
</Class>
```

Also consider the following template:

```
<?xml version="1.0"?>  
<!DOCTYPE _TDEBlock_ SYSTEM "dtd/tde.dtd">  
<_TDEBlock_ DOMTree="class.xml">  
  <define MACRO=newMethodName>addNew</define>  
    <targetscope NAME="Method">  
      <updatetargetscope NAME="name"  
TYPE="ATTRIBUTE">$newMethodName$</addtargetscope>  
      </targetscope>  
</_TDEBlock_>
```

the result of applying the template is as follows:

```
<Class name="Set">  
<Method name="addNew">  
</Method>  
5 <Method name="del">  
</Method>  
</Class>
```

Notice the name of the first method changes from "add" to "addNew".

10 removetargetscope directive

The removetargetscope directive is used to delete an element from a DOM tree associated with a domain model.

Syntax

```
<removetargetscope NAME="sname" INDEX=_index_/>
```

15 where,

```
    sname = target scope name  
    _index_ = integer value
```

Description

20 The sname specifies the name of the scope. The _index_ value specifies which element to remove based on an index value. The default value is set to 0. Therefore, if there is more than one element, the first element is removed. If the index is set to -1 the last child element is removed.

Example

Suppose the model defines an element 'Class' which has two other embedded elements. Lets say that this document is called class.xml

```
<Class name="Set">  
5 <Method name="add">  
  </Method>  
  <Method name="del">  
    </Method>  
  </Class>
```

10

Also consider the following template:

```
<?xml version="1.0"?>  
<!DOCTYPE _TDEBlock_ SYSTEM "dtd/tde.dtd">  
<_TDEBlock_ DOMTree="class.xml">  
15   <removetargetscope NAME="Method"/>  
</_TDEBlock_>
```

15

the result of applying the template is as follows:

```
<Class name="Set">  
<Method name="del">  
20 </Method>  
</Class>
```

20

Note that the first method element was removed. Since no index was specified the default value for the index is 0.

updatetargetdoctype directive

The `updatetargetdoctype` directive is used to update the !DOCTYPE element of a DOM tree associated with a domain model. If the element does not exist is it inserted into the DOM Tree.

5 Syntax

```
<updatetargetdoctype SYSTEM_URL="SYSTEM_URL"  
PUBLIC_NAME="DTD_NAME" PUBLIC_URL="DTD_URL"  
ROOT_ELEMENT_NAME="root_element">
```

where,

- 10 DTD_name = a name that identifies the dtd file
- DTD_URL = an absolute or relative url where the dtd file
can be found
- SYSTEM_URL = an absolute or relative url where the dtd
file can be found
- 15 root_element = the root element tag name

Description

- The DTD_URL is an absolute or relative path where the dtd file can be found. The DTD_URL is used if one wants to specify a public dtd. This attribute should only be
- 20 specified if a public dtd is being used. Similarly the
SYSTEM_URL is also an absolute or relative path where the
dtd file can be found. This should be specified if a system
dtd is being used. The DTD_name is a identifier that
identifies the dtd file and is used when specifying a public
 - 25 dtd file. The root_element is the name of the root element
of the document. If the root_element is not specified the
mechanism will search the target document and find the root
element tag name.

Example1: Adding a simple !DOCTYPE element

Consider the following template file:

```
<?xml version="1.0"?>
<!DOCTYPE _TDEBlock_ SYSTEM "../../../dtd/tde.dtd">
5 <_TDEBlock_>
    <updatetargetdoctype PUBLIC_NAME="Chemistry"
PUBLIC_URL="http://sunsite/unc.edu/public/chemistry.dtd"
ROOT_ELEMENT_NAME="myroot"/>
</_TDEBlock_>
```

the result of applying the template is as follows:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE myroot PUBLIC "Chemistry"
"http://sunsite/unc.edu/public/chemistry.dtd">
```

Example2: Adding a simple !DOCTYPE element plus an element.

Consider the following template:

```
<?xml version="1.0"?>
<!DOCTYPE _TDEBlock_ SYSTEM "../../../dtd/tde.dtd">
<_TDEBlock_>
20 <updatetargetdoctype SYSTEM_URL="mydtd.dtd"
ROOT_ELEMENT_NAME="MyMember"/>
    <updatetargetscope NAME="MyMember">
        This is the text for the new member
    </updatetargetscope>
25 </_TDEBlock_>
```


the result of applying the template is as follows:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE MyMember SYSTEM "mydtd.dtd">
<MyMember/>
```

- 5 Example3: Adding a simple !DOCTYPE element after adding an element.

Consider the following template:

93230"0T0H960

```
<?xml version="1.0"?>
<!DOCTYPE _TDEBlock_ SYSTEM "../.../.../dtd/tde.dtd">
10 <_TDEBlock_
    <updatetargetscope NAME="MyMember">
        This is the text for the new member
    </updatetargetscope>
    <updatetargetdoctype SYSTEM_URL="mydtd.dtd"/>
15 </_TDEBlock_>
```

the result of applying the template is as follows:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE MyMember SYSTEM "mydtd.dtd">
<MyMember>
20     This is the text for the new member
</MyMember>
```

- Note that since the root node already exists (ie MyMember is created) at the point where "updatetargetdoctype" is defined, the ROOT_ELEMENT attribute does not need to be
- 25 specified. The mechanism will know that "MyMember" is the

root element and fill in the appropriate information in the !DOCTYPE element.

addattribute directive

The addattribute directive adds an attribute to an existing
5 element.

Syntax

<addattribute NAME="attrname">attribute value</addattribute>

where,

attrname = attribute name.

10 Description

The attrname specifies the name of the attribute. If the attribute already exists the attribute will not be added to the element.

Example

15 Suppose the model defines an element 'Class' which has two other embedded elements. This document is called class.xml.

<Class name="Set">

<Method name="add">

</Method>

20 <Method name="del">

</Method>

</Class>

Also consider the following template:

```

<?xml version="1.0"?>
<!DOCTYPE _TDEBlock_ SYSTEM "dtd/tde.dtd">
<_TDEBlock_ DOMTree="class.xml">
<targetscope NAME="Class">
5   <addtargetscope NAME="Method">
      <addattribute NAME="name">newMethod</addattribute>
    </addtargetscope>
  </targetscope>
</_TDEBlock_>

```

10 the result of applying the template is as follows:

```

<Class name="Set">
<Method name="newMethod">
</Method>
<Method name="add">
15 </Method>
<Method name="del">
</Method>
</Class>

```

20 updateattribute directive

The updateattribute directive behaves in the same manner as the addattribute directive.

Syntax

```
<addattribute NAME="attrname">attribute value</addattribute>
```

25 where,

attrname = attribute name.

Description

The attrname specifies the name of the attribute. If the attribute already exists the attribute value will be updated with the value specified in this directive.

5 Example

Suppose the model defines an element ``Class'' which has two other embedded elements. This document is called class.xml.

document "class.xml"

```
10 <Class name="Set">
    <Method name="add">
    </Method>
    <Method name="del">
    </Method>
    </Class>
```

15 Also consider the following template:

```
20 <?xml version="1.0"?>
    <!DOCTYPE _TDEBlock_ SYSTEM "dtd/tde.dtd">
    <_TDEBlock_ DOMTree="class.xml">
    <targetscope NAME="Class">
    <addtargetscope NAME="Method">
    <updateattribute NAME="name">newMethod</addattribute>
    </addtargetscope>
    <updateattribute NAME="name">NewSetClass</addattribute>
    </targetscope>
25 </_TDEBlock_>
```

the result of applying the template is as follows:

```

<Class name="NewSetClass">
  <Method name="newMethod">
  </Method>
  <Method name="add">
5  </Method>
  <Method name="del">
  </Method>
</Class>

```

10 Note the first updateattribute directive adds an attribute named "newMethod" on the newly created method element. The second updateattribute directive updates the "name" attribute on the Class element.

removeattribute directive

15 The removeattribute directive removes an attribute from an existing element.

Syntax

```
<removeattribute NAME="attrname"/>
```

where,

attrname = target attribute name

20 Description

The attrname specifies the name of the attribute.

Example

Suppose the model defines an element ``Class'' which has two other embedded elements. This document is called class.xml.

CA919990037US1

```
<Class name="Set">
<Method name="add">
</Method>
<Method name="del">
5 </Method>
</Class>
```

Also consider the following template:

```
10 <?xml version="1.0"?>
<!DOCTYPE _TDEBlock_ SYSTEM "dtd/tde.dtd">
<_TDEBlock_ DOMTree="class.xml">
<targetscope NAME="Method">
    <removeattribute NAME="name"/>
</targetscope>
15 </_TDEBlock_>
```

the result of applying the template is as follows:

```
<Class name="Set">
<Method>
</Method>
20 <Method name="del">
</Method>
</Class>
```

Note the "name" attribute on the first Method element was removed.

25

CA919990037US1

addtext directive

The addtext directive adds text to an existing element.

Syntax

```
<addtext INDEX="_index_">text content</addtext>
```

5 where,

index = integer value

Description

10 The _index_ value specifies where to insert the text. The default value of the index is set to 0. Therefore, the text content that is added will become the first child element.

Example

Suppose the model defines an element 'Class' which has two other embedded elements. This document is called class.xml.

15 <Class name="Set">
<Method name="add">
</Method>
<Method name="del">
</Method>
</Class>

20

Also consider the following template:

25 <?xml version="1.0"?>
<!DOCTYPE _TDEBlock_ SYSTEM "dtd/tde.dtd">
<_TDEBlock_ DOMTree="class.xml">
<targetscope NAME="Class">

```

    <addtext>This class is a collection of objects</addtext>
</targetscope>
</_TDEBlock_>

```

the result of applying the template is as follows:

```

5  <Class name="Set">
    This class is a collection of objects
    <Method name="add">
        </Method>
    <Method name="del">
10  </Method>
    </Class>

```

updatetext directive

```

15 The updatetext directive behaves in the same manner as the
    addtext directive.

```

Syntax

```

<updatetext INDEX="_index_">text content</updatetext>

```

where,

```

    _index_ = integer value

```

```

20 Description

```

The `_index_` value specifies which text content to update.

The default value of the index is set to 0. Therefore, the text content that is updated will become the first child element. If no text content is found at the specified index

```

25 it is added at the index specified.

```


Example

Suppose the model defines an element ''Class'' which has two other embedded elements. This document is called class.xml.

```
<Class name="Set">
5  This is a collection of objects
  <Method name="add">
    </Method>
    <Method name="del">
      </Method>
10 </Class>
```

Also consider the following template:

```
<?xml version="1.0"?>
<!DOCTYPE _TDEBlock_ SYSTEM "dtd/tde.dtd">
15 <_TDEBlock_ DOMTree="class.xml">
  <targetscope NAME="Class">
    <targetscope NAME="Method">
      <updatetext>//This method adds an object to the
collection</updatetext>
20    </targetscope>
      <updatetext>//This is a collection of objects. Also known as
a set of objects.</updatetext>
    </targetscope>
  </_TDEBlock_>
```

25 the result of applying the template is as follows:

```
<Class name="NewSetClass">
```

//This is a collection of objects. Also known as a set of objects.

```
<Method name="add">
```

```
//This method adds an object to the collection
```

5 </Method>

```
<Method name="del">
```

```
</Method>
```

```
</Class>
```

10 Note the first updatetext directive adds text content to the first method element. The second updatetext directive updates text content associated with the Class element.

removetext directive

The removetext directive removes text content from an existing element.

15 Syntax

```
<removetext INDEX="_index_"/>
```

where,

```
    _index_ = integer value
```

Description

20 The _index_ value specifies which text content to remove. The default value of the index is set to 0. Therefore, the text content that is removed will be the first child element.

Example

Suppose the model defines an element "Class" which has two other embedded elements. This document is called class.xml.

```
<Class name="Set">
```

This is a collection of objects

```
5 <Method name="add">
```

```
</Method>
```

```
<Method name="del">
```

```
</Method>
```

```
</Class>
```

10

Also consider the following template:

```
<?xml version="1.0"?>
```

```
<!DOCTYPE _TDEBlock_ SYSTEM "dtd/tde.dtd">
```

```
<_TDEBlock_ DOMTree="class.xml">
```

```
15 <removetext/>
```

```
</_TDEBlock_>
```

the result of applying the template is as follows:

```
<Class name="Set">
```

```
<Method>
```

```
20 </Method>
```

```
<Method name="del">
```

```
</Method>
```

```
</Class>
```

Note the text content under the Class element was removed.

25 hastargetscope directive

The hastargetscope directive behave in a similar manner as the hasscope directive. However, this directive applies to the targeted DOM tree. If the targeted DOM tree contains the scope name the section of code defined within this
5 directive is parsed.

Syntax

<hastargetscope NAME="sname" INDEX=_index_>
section
</hastargetscope>
10 where,
 sname = target scope name
 index = integer value

Description

The sname specifies the name of the scope. The _index_
15 value specifies if the element at the index value exists.
The default value is set to 0.

Example

Suppose the model defines an element ''Class'' which has two other embedded elements. This document is called class.xml.
20 <Class name="Set">
 <Method name="add">
 </Method>
 <Method name="del">
 </Method>
25 </Class>

Also consider the following template:

```

<?xml version="1.0"?>
<!DOCTYPE _TDEBlock_ SYSTEM "dtd/tde.dtd">
<_TDEBlock_ DOMTree="class.xml">
<targetscope NAME="Class">
5     <hastargetscope NAME="NOElement"/>
        <removetargetscope NAME="Method"/>
        </hastargetscope>
</targetscope>
</_TDEBlock_>

```

10 this resulting DOM tree is as follows:

```

<Class name="Set">
<Method name="add">
</Method>
<Method name="del">
15 </Method>
</Class>

```

Note that no changes were made. Since the "NOElement" was not found in the targeted DOM tree everything within the hastargetscope does not get processed.

20 repeattargetscope directive

The repeattargetscope directive behaves in a similar manner as the repeatscope directive. However, this directive applies to the targeted DOM tree. The repeattargetscope iterates over a list of child elements.

Syntax

```
<repeattargetscope NAME="sname">
  section
</repeattargetscope>
```

- 5 where,
- sname = target scope name

Description

The sname specifies the name of the scope.

Example

- 10 Suppose the model defines an element "Class" which has two
 other embedded elements. This document is called class.xml.

```
<Class name="Set">
  <Method name="add">
    </Method>
15   <Method name="del">
    </Method>
  </Class>
```

Also consider the following template:

```
20   <?xml version="1.0"?>
      <!DOCTYPE _TDEBlock_ SYSTEM "dtd/tde.dtd">
      <_TDEBlock_ DOMTree="class.xml">
        <repeattargetscope NAME="Method">
          <addattribute NAME="type">void</addattribute>
25   </repeattargetscope>
      </_TDEBlock_>
```

this resulting DOM tree is as follows:

```
<Class name="Set">  
<Method name="add" type="void">  
</Method>  
5 <Method name="del" type="void">  
</Method>  
</Class>
```

ifhasrepeattargetscope directive

- 10 The ifhasrepeattargetscope directive behaves in a similar manner as the ifhasrepeatscope directive. However, this directive applies to the targeted DOM tree. The ifhasrepeattargetscope directive iterates over a list of child elements. If a child list does not exist the section
- 15 of text between the directive does not get processed. This directive may also be used with the else statement.

Syntax

```
<ifhasrepeattargetscope NAME="sname">  
sectionA  
20 <else/>  
sectionB  
</ifhasrepeattargetscope>
```

where,

sname = target scope name

25 Description

The sname specifies the name of the scope.

Example

Suppose the model defines a list of elements. This document is called element.xml.

```
<ElementList>
```

```
5  <Element index="1"/>
```

```
<Element index="2"/>
```

```
<Element index="3"/>
```

```
<Element index="4"/>
```

```
<Element index="5"/>
```

```
10 <Element index="6"/>
```

```
<Element index="7"/>
```

```
<Element index="8"/>
```

```
</ElementList>
```

```
15 Also consider the following template:
```

```
<?xml version="1.0"?>
```

```
<!DOCTYPE _TDEBlock_ SYSTEM "dtd/tde.dtd">
```

```
<_TDEBlock_ DOMTree="element.xml">
```

```
    <ifhasrepeattargetscope NAME="NOElement">
```

```
20      <addattribute NAME="name">ElementName</addattribute>
```

```
    <else/>
```

```
      <addtargetscope NAME="NOElement"/>
```

```
    </ifhasrepeattargetscope>
```

```
</_TDEBlock_>
```

```
25 this resulting DOM tree is as follows:
```

```
<ElementList>
```

```
<NOElement>
```


where,

sname = target scope name

index = integer value

Description

- 5 The sname specifies the name of the scope. The _index_ value specifies if the element at the index value exists. The default value is set to 0.

Example

- 10 Suppose the model defines an element "Class" which has two other embedded elements. This document is called class.xml.

```
<Class name="Set">
  <Method name="add">
  </Method>
  <Method name="del">
15 </Method>
</Class>
```

Also consider the following template:

```
<?xml version="1.0"?>
20 <!DOCTYPE _TDEBlock_ SYSTEM "dtd/tde.dtd">
  <_TDEBlock_ DOMTree="class.xml">
    <targetscope NAME="Class">
      <ifhastargetscope NAME="Method"/>
        <addattribute NAME="type">void</addattribute>
25 <else/>
      <addtargetscope NAME="Method"/>
    </ifhastargetscope>
```

```

    <ifhastargetscope NAME="NoMethod"/>
        <addtargetscope NAME="Element"></addtargetscope>
    <else/>
        <addtargetscope NAME="newMethod"/>
5    </ifhastargetscope>
</targetscope>
</_TDEBlock_>

```

this resulting DOM tree is as follows:

```

10 <Class name="Set">
    <Method name="add" type="void">
    </Method>
    <Method name="del">
    </Method>
    <newmethod>
15 </newmethod>
    </Class>

```

Note that the attribute was added to the first method. No element changes were made. Since the "NoMethod" was not found in the targeted DOM tree, the else statement gets processed and a new element is added called newmethod.

createalias directive

The createalias directive is used to define logical roots. These logical roots acts as pointers on a DOM tree.

25 Syntax

```
<createalias ALIASNAME="alias" ALIASPATH="sname"/>
```

CA919990037US1

where,

sname = target scope name

alias = the alias name that represents the logical root

Description

- 5 The sname specifies the name of the scope. The alias represents the logical root. Using alias names simplifies navigating a DOM tree. Furthermore, using alias names can make the template more readable. Alias names are globally visible just like macros.

10 Example

Consider the following source XML document:

```
<Package name=MyPackage>
  <ClassList>
    <Class name=HondaCar>
      <Method name=addHondaCar>
15      </Method>
      <Method name=removeHondaCar>
      </Method>
    </Class>
20    <Class name=FordCar>
      <Method name=addFordCar>
      </Method>
      <Method name=removeFordCar>
      </Method>
25    </Class>
  </ClassList>
</Package>
```

Now consider the following template file:

```
<_TDEBlock_>
  <createalias ALIASNAME="HondaClass"
ALIASPATH="ClassList//Class[0]"/>
5   <scope NAME="ClassList//Class[1]">
    <createalias ALIASNAME="FordAddCarMethod"
ALIASPATH="Method[0]"/>
    </scope>
    <scope NAME="HondaClass">
10      <repeatscope NAME="Method">
        $name$
      </repeatscope>
    </scope>
    <scope NAME="ClassList//Class[1]">
15      $FordAddCarMethod//name$
    </scope>
  </_TDEBlock_>
```

The resulting generated output is as follows:

```
addHondaCar
20  removeHondaCar
    addFordCar
```

The above example defines two alias names. The alias name "HondaClass" defines a logical root that references the first class element in the DOM tree. The second alias, 25 "FordAddCarMethod", references the first method under the second class. As one can see alias names can be used to

Text Content

Description

An XML document consists of child elements that may or may not have a text content. The text content of a child
5 element is represented by a macro called \$TEXT\$. Consider the following XML document.

```
<class>
  <Name>Foo</Name>
</class>
```

- 10 A template file can access the text contents of the 'Name' element as follows (see Scoping Directive Section):

```
<_TDEBlock_>
<scope NAME="class">
  <scope NAME="Name">
15     $TEXT$
  </scope>
</scope>
</_TDEBlock_>
```

The resulting emitted code would be:

20 Foo

The template file can get complicated if numerous text contents need to be accessed. A scoping rule for each sibling would need to be implemented. In order to improve usability a scoping operator can be used. This scoping
25 operator is defined as '//'. Therefore the above template file can be rewritten as follows:

CA919990037US1

If no index is given the first sibling that matches the macro is used. Similarly, each element can have a list of text contents. Consider the following XML document.

```
<class>
```

```
5      Extra Text1
      <Name>Foo</Name>
      Extra Text2
      <Method>MyMethod</Method>
      Extra Text3
```

```
10 </class>
```

Each text content under an element is given an index value. This index identifies each text content so that positional navigation can be used to navigate the text content list. This is illustrated in the following template file.

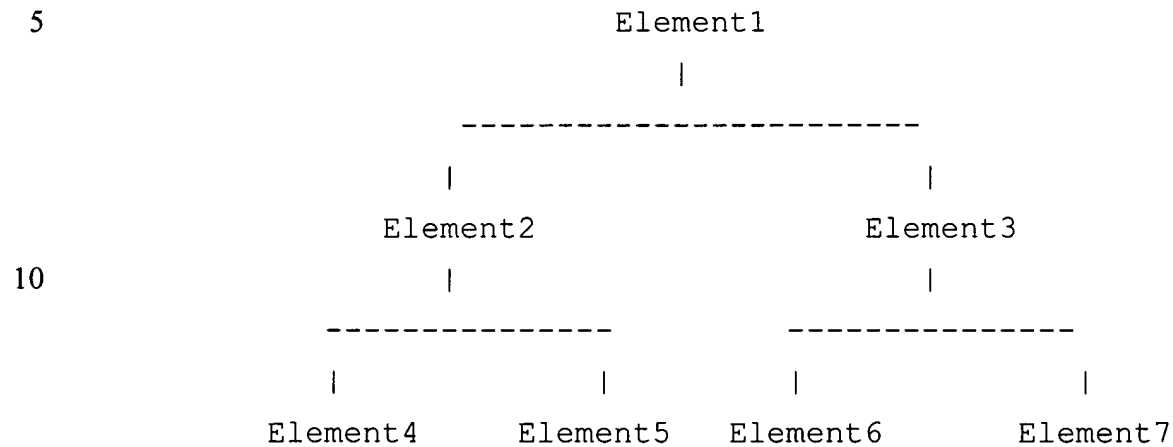
```
15 <_TDEBlock_>
    <scope NAME="class">
        Name=$Name//Text$
        First Text =$Text[1]$
        Second Text =$Text[2]$
20      Third Text =$Text[3]$
    </scope>
</_TDEBlock_>
```

The resulting output is emitted:

```
25      Name=Foo
      First Text =Extra Text1
      Second Text =Extra Text2
      Third Text =Extra Text3
```


Children Navigation

The mechanism provides means to navigate an elements children nodes generically. For example consider the following tree:



One might want to navigate to all the end nodes of the tree without the hassle of knowing Element2 and Element3. Or one might want to repeat on all the children nodes of an element. Typically, in order to process Element2 and Element3 a separate scoping directive would have to be written for each element. For example consider the following template file:

15

20

```
<_TDEBlock_>
  <scope NAME="Element2">
    $Name$
  </scope>
  <scope NAME="Element3">
    $Name$
  </scope>
</_TDEBlock_>
```

25

Note for both Elements we are extracting the Name attribute. The template file can be huge if there were more children elements. Just like there is a predefined string to represent a parent for an element (ie. "..") there should be
5 a predefined string to represent the children elements (ie. _TDECHILDREN_). With this in mind the above template can be rewritten as follows:

<_TDEBlock_>
 <repeatscope NAME="_TDECHILDREN_">
10 \$Name\$
 </repeatscope>
<_TDEBlock_>

One can also navigate to a specific child node based on index. For example consider the following template file:

15 <_TDEBlock_>
 <scope NAME="_TDECHILDREN_[1]">
 \$Name\$
 </scope>
<_TDEBlock_>

20

Tag Name Content

Description

The mechanism provides a predefined macro to get the tag name of an element. This macro is called `$_TDETAGNAME_$`.

5 Consider the following XML document.

```
<class>
  <Name>Foo</Name>
</class>
```

10 A template file can access the tag name content of the 'Name' element as follows:

```
<_TDEBlock_>
<scope NAME="class">
  <scope NAME="Name">
    $_TDETAGNAME_$
  </scope>
</scope>
</_TDEBlock_>
```

15

The resulting emitted code would be:

Name

20 Consider a list of students that each have a list of marks as represented by the following XML document:

```
<student_list>
  <student>
    <id>166443</id>
    <name>Sean Watt</name>
    <marks>
```

25

Suppose all the biology marks and math marks are desired.
We can write the following template to extract this
information from the above document:

```

5      <_TDEBlock_>
      <scope NAME="student_list">
          <repeatscope NAME="student">
              <scope NAME="marks">
                  <repeatscope NAME="_TDECHILDREN_">
                      <if
10      EXPRESSION="(($_TDETAGNAME_$=="biology";) |
        ($_TDETAGNAME_$=="math";))">
                                $TEXT$
                      </if>
                  </repeatscope>
          </scope>
15      </repeatscope>
      </scope>
</_TDEBlock_>

```

20 The above template file navigates down to the mark element
and then navigates through all the children elements under
the mark element. An if statement matches the tag name of
each child element. If any of the child elements are equal
to "math" or "biology" the text content of the child element
is extracted.

25 The resulting emitted code is as follows:

	63
	75
	88
	78
5	73
	92

Comments

Comments help makes the template files easier to read. They do not get emitted in the output file. A blank line on the other hand will get emitted in the output file.

Syntax

```
<!-- comment -->
```

Example

The following are some comment lines in the template file.

```
<!--
```

```
  This is a template file
```

```
-->
```